TURING 图灵程序设计丛书

[PACKT] PUBLISHING



[美] Prateek Joshi 著 陶俊杰 陈小莉 译

Python 机器学习经典实例

Python Machine Learning Cookbook

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

作者简介

Prateek Joshi

人工智能专家，重点关注基于内容的分析和深度学习，曾在英伟达、微软研究院、高通公司以及硅谷的几家早期创业公司任职。

TURING 图灵程序设计丛书

Python 机器学习经典实例

Python Machine Learning Cookbook

[美] Prateek Joshi 著　陶俊杰　陈小莉 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Python机器学习经典实例 / (美) 普拉提克·乔西(Prateek Joshi) 著 ; 陶俊杰, 陈小莉译. -- 北京 : 人民邮电出版社, 2017.8
（图灵程序设计丛书）
ISBN 978-7-115-46527-6

Ⅰ. ①P… Ⅱ. ①普… ②陶… ③陈… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2017)第178314号

内 容 提 要

在如今这个处处以数据驱动的世界中，机器学习正变得越来越大众化。它已经被广泛地应用于不同领域，如搜索引擎、机器人、无人驾驶汽车等。本书首先通过实用的案例介绍机器学习的基础知识，然后介绍一些稍微复杂的机器学习算法，例如支持向量机、极端随机森林、隐马尔可夫模型、条件随机场、深度神经网络，等等。

本书是为想用机器学习算法开发应用程序的 Python 程序员准备的。它适合 Python 初学者阅读，不过熟悉 Python 编程方法对体验示例代码大有裨益。

◆ 著　[美] Prateek Joshi
译　陶俊杰　陈小莉
责任编辑　朱　巍
执行编辑　徐晓娟
责任印制　彭志环
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷
◆ 开本：800×1000　1/16
印张：16.5
字数：390千字　2017年 8 月第 1 版
印数：1－35 00册　2017年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-9523号

定价：59.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

译者序

有一天，忽然想到自己整天面对着52个英文字母、9个数字、32个符号[①]和一个空格，经常加班没有双休日，好傻。时间不断被各种噪声碎片化，完全就是毛姆在《月亮和六便士》里写的，"If you look on the ground in search of a sixpence, you don't look up, and so miss the moon"，整天低头刷手机，却不记得举头望明月。生活也愈发无序，感觉渐渐被掏空。薛定谔的《生命是什么》给我提了个醒，他在"以'负熵'为生"（It Feeds On 'negative Entropy'）一节指出："要活着，唯一的办法就是从环境里不断地汲取负熵。"在介绍了熵的概念及其统计学意义之后，他紧接着在"从环境中引出'有序'以维持组织"（Organization Maintained By Extracting 'Order' From The Environment）一节进一步总结："一个有机体使本身稳定在较高的有序水平上（等于熵的相当低的水平上）的办法，就是从环境中不断地吸取秩序。"这个秩序（负熵、$k\log(1/n)$）可以是食物，也可以是知识，按主流叫法就是"正能量"（有些所谓正能量却碰巧是增加系统无序水平的正熵）。于是，我开始渐渐放弃那些让人沮丧的老梗，远离那些引发混乱的噪声，重新读书，试着翻译，学会去爱。这几年最大的收获就是明白了"隔行如隔山"的道理，试着循序渐进，教学相长，做力所能及之事，让编程变简单。

一般人都不喜欢编程，更不喜欢动手编程（时间消耗：编写 & 测试 40%、重构 40%、风格 & 文档 20%），却喜欢在心里、嘴上编程："先这样，再那样，如果要XX，就YY，最后就可以ZZ了。"分分钟就可以说完几万行代码的项目，水还剩大半杯。一旦大期将近，即使要亲自动手Copy代码，也会觉得苦堪搬砖，键盘不是红与黑、屏幕不能左右推、小狗总是闹跑追，不断在数不清的理由中增加自己的熵。偶尔看编程书的目的也很明确，就是为了快速上手，找到答案。当然也是在Google、StackOverflow、GitHub网站上找不到答案之后，无可奈何之举。编程书把看着复杂的知识写得更复杂，虽然大多篇幅不输"飞雪连天射白鹿，笑书神侠倚碧鸳"等经典，且纲举目张、图文并茂，甚至有作者爱引经据典，却极少有令人拍案的惊奇之处。为什么同样是文以载道，编程书却不能像武侠小说一样简单具体，反而显得了无生趣，令人望而却步？虽然编程的目的就是用计算机系统解决问题，但是大多数问题的知识都在其他领域中，许多作者在介绍编程技巧时，又试图介绍一些并不熟悉的背景知识，显得生涩难懂，且增加了书的厚度。

---

① 见文末Python示例代码。

有时我们真正需要的，就是能快刀斩乱麻的代码。（Talk is cheap, show me the code.）编程与研究数理化不同，没有任何假设、原命题、思维实验，并非科学；与舞剑、奏乐、炒菜相似，都是手艺，只要基础扎实，便结果立判。编程技巧也可以像剑谱、乐谱、食谱一般立竿见影，这本《Python机器学习经典实例》正是如此，直接上代码，照着做就行，不用纠结为什么。

机器学习是交叉学科，应用广泛，目前主流方法为统计机器学习。既然是以统计学为基础，那么就不只是计算机与数学专业的私房菜了，机器学习在自然科学、农业科学、医药科学、工程与技术科学、人文与社会科学等多种学科中均可应用。如果你遇到了回归、分类、预测、聚类、文本分析、语音识别、图像处理等经典问题，需要快速用Python解决，那么这本菜谱适合你。即使你对机器学习方法还一知半解，也不妨一试。毕竟是Python的机器学习，还能难到哪儿去呢？目前十分流行的Python机器学习库scikit-learn①是全书主角之一，功能全面，接口友好，许多经典的数据集和机器学习案例都来自Kaggle②。若有时间追根溯源，请研究周志华教授的《机器学习》西瓜书，周教授啃着西瓜把机器学习调侃得淋漓尽致，详细的参考文献尤为珍贵。但是想当作菜谱看，拿来就用，还是需要费一番功夫；若看书不过瘾，还有吴恩达（Andrew Ng）教授在Coursera上的机器学习公开课③，机器学习入门最佳视频教程，吴教授用的工具是Matlab的免费开源版本Octave④，你也可以用Python版⑤演示教学示例。

学而时习之，不亦乐乎。学习编程技巧，解决实际问题，是一件快乐的事情。希望这本Python机器学习经典案例，可以成为你的负熵，帮你轻松化解那些陈年老梗。如果再努努力，也许陆汝钤院士在《机器学习》序言中提出的6个问题⑥，你也有答案了。

示例代码：

```
"""打印ASCII字母表、数字、标点符号"""

import string

for item in [string.ascii_letters,
```

① scikit-learn网址：http://scikit-learn.org/stable/。

② Kaggle是一个2010年成立的数据建模和数据分析竞赛平台，全球数据科学家、统计学家、机器学习工程师的聚集地，上面有丰富的数据集，经典的机器学习基础教程，以及让人流口水的竞赛奖金，支持Python、R、Julia、SQLite，同时也支持jupyter notebook在线编程环境，2017年3月8日被谷歌收购。

③ 分免费版和付费版（购买结业证书），学习内容一样，https://zh.coursera.org/learn/machine-learning。

④ Octave下载地址：https://www.gnu.org/software/octave/。

⑤ GitHub项目：https://github.com/mstampfer/Coursera-Stanford-ML-Python。

⑥ 陆院士的6个问题分别是：1. 机器学习早期的符号机器学习，如何在统计机器学习主流中发展；2. 统计机器学习算法中并不现实的“独立同分布”假设如何解决；3. 深度学习得益于硬件革命，是否会取代统计机器学习；4. 机器学习用的都是经典的概率统计、代数逻辑，而目前仅有倒向微分方程用于预测，微分几何的流形用于降维（流形学习，Manifold learning，科普见博文http://blog.pluskid.org/?p=533），只是数学领域的一角，其他现代数学理论是否可以参与其中；5. 机器学习方法仍不够严谨，例如目前流形学习直接将高维数据集假设成微分流形，需要进一步完善；6. 大数据与统计机器学习是如何互动的。

```
            string.digits,
            string.punctuation]:
    print('{}\t{}'.format(len(item), item))
```

输出结果：

```
52 abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
10 0123456789
32 !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

前　　言

在如今这个处处以数据驱动的世界中，机器学习正变得越来越大众化。它已经被广泛地应用于不同领域，如搜索引擎、机器人、无人驾驶汽车等。本书不仅可以帮你了解现实生活中机器学习的应用场景，而且通过有趣的菜谱式教程教你掌握处理具体问题的算法。

本书首先通过实用的案例介绍机器学习的基础知识，然后介绍一些稍微复杂的机器学习算法，例如支持向量机、极端随机森林、隐马尔可夫模型、条件随机场、深度神经网络，等等。本书是为想用机器学习算法开发应用程序的Python程序员准备的。它不仅适合Python初学者（当然，熟悉Python编程方法将有助于体验示例代码），而且也适合想要掌握机器学习技术的Python老手。

通过本书，你不仅可以学会如何做出合理的决策，为自己选择合适的算法类型，而且可以学会如何高效地实现算法以获得最佳学习效果。如果你在图像、文字、语音或其他形式的数据处理中遇到困难，书中处理这些数据的机器学习技术一定会对你有所帮助！

本书内容

第1章介绍各种回归分析的监督学习技术。我们将学习如何分析共享自行车的使用模式，以及如何预测房价。

第2章介绍各种数据分类的监督学习技术。我们将学习如何评估收入层级，以及如何通过特征评估一辆二手汽车的质量。

第3章论述支持向量机的预测建模技术。我们将学习如何使用这些技术预测建筑物里事件发生的概率，以及体育场周边道路的交通情况。

第4章阐述无监督学习算法，包括K-means聚类和均值漂移聚类。我们将学习如何将这些算法应用于股票市场数据和客户细分。

第5章介绍推荐引擎的相关算法。我们将学习如何应用这些算法实现协同滤波和电影推荐。

第6章阐述与文本数据分析相关的技术，包括分词、词干提取、词库模型等。我们将学习如何使用这些技术进行文本情感分析和主题建模。

第7章介绍与语音数据分析相关的算法。我们将学习如何建立语音识别系统。

第8章介绍分析时间序列和有序数据的相关技术，包括隐马尔可夫模型和条件随机场。我们将学习如何将这些技术应用到文本序列分析和股市预测中。

第9章介绍图像内容分析与物体识别方面的算法。我们将学习如何提取图像特征，以及建立物体识别系统。

第10章介绍在图像和视频中检测与识别面部的相关技术。我们将学习使用降维算法建立面部识别器。

第11章介绍建立深度神经网络所需的算法。我们将学习如何使用神经网络建立光学文字识别系统。

第12章介绍机器学习使用的数据可视化技术。我们将学习如何创建不同类型的图形和图表。

阅读背景

Python 2.*x*和Python 3.*x*的版本之争尚未平息①。一方面，我们坚信世界会向更好的版本不断进化，另一方面，许多开发者仍然喜欢使用Python 2.*x*的版本。目前许多操作系统仍然内置Python 2.*x*。本书的重点是介绍Python机器学习，而非Python语言本身。另外，考虑到程序的兼容性，书中用到了一些尚未被迁移到Python 3.*x*版本的程序库，因此，本书依然选择Python 2.*x*的版本。我们会尽最大努力保持代码兼容各种Python版本，因为这样可以让你轻松地理解代码，并且很方便地将代码应用到不同场景中。

读者对象

本书是为想用机器学习算法开发应用程序的Python程序员准备的。它适合Python初学者阅读，不过熟悉Python编程方法对体验示例代码大有裨益。

内容组织

在本书中，你会频繁地看到下面这些标题（准备工作、详细步骤、工作原理、更多内容、另请参阅）。

为了更好地呈现内容，本书采用以下组织形式。

① 2020年之前应该不会终结。——译者注

准备工作

这部分首先介绍本节目标，然后介绍软件配置方法以及所需的准备工作。

详细步骤

这部分介绍具体的实践步骤。

工作原理

这部分通常是对前一部分内容的详细解释。

更多内容

这部分会补充介绍一些信息，帮助你更好地理解前面的内容。

另请参阅

这部分提供一些参考资料。

排版约定

在本书中，你会发现一些不同的文本样式。这里举例说明它们的含义。

嵌入代码、命令、选项、参数、函数、字段、属性、语句等，用等宽的代码字体显示："这里，我们将25%的数据用于测试，可以通过`test_size`参数进行设置。"

代码块用如下格式：

```
import numpy as np
import matplotlib.pyplot as plt

import utilities

# Load input data
input_file = 'data_multivar.txt'
X, y = utilities.load_data(input_file)
```

命令行输入或输出用如下格式：

```
$ python object_recognizer.py --input-image imagefile.jpg --model-file
erf.pkl --codebook-file codebook.pkl
```

新术语和重要文字将采用黑体字。你在屏幕上看到的内容，包括对话框或菜单里的文本，都将这样显示："如果你将数组改为`(0, 0.2, 0, 0, 0)`，那么**Strawberry**部分就会高亮显示。"

读者反馈

我们非常欢迎读者的反馈。如果你对本书有些想法，有什么喜欢或是不喜欢的，请反馈给我们，这将有助于我们出版充分满足读者需求的图书。

一般性反馈请发送电子邮件至feedback@packtpub.com，并在邮件主题中注明书名。

如果你在某个领域有专长，并有意编写一本书或是贡献一份力量，请参考我们的作者指南，地址为http://www.packtpub.com/authors。

客户支持

你现在已经是引以为傲的Packt读者了。为了能让你的购买物超所值，我们还为你准备了以下内容。

下载示例代码

你可以用你的账户从http://www.packtpub.com下载所有已购买Packt图书的示例代码文件。如果你是从其他途径购买的本书，可以访问http://www.packtpub.com/support并注册，我们将通过电子邮件把文件发送给你。

可以通过以下步骤下载示例代码文件：

(1) 用你的电子邮件和密码登录或注册我们的网站；
(2) 将鼠标移到网站上方的**客户支持**（SUPPORT）标签；
(3) 单击**代码下载与勘误**（Code Downloads & Errata）按钮；
(4) 在**搜索框**（Search）中输入书名；
(5) 选择你要下载代码文件的书；
(6) 从下拉菜单中选择你的购书途径；
(7) 单击**代码下载**（Code Download）按钮。

你也可以通过单击Packt网站上本书网页上的**代码文件**（Code Files）按钮来下载示例代码，该网页可以通过在**搜索框**（Search）中输入书名获得。以上操作的前提是你已经登录了Packt网站。

下载文件后，请确保用以下软件的最新版来解压文件：

- WinRAR / 7-Zip for Windows；
- Zipeg / iZip / UnRarX for Mac；
- 7-Zip / PeaZip for Linux 。

本书的代码包也可以在GitHub上获得，网址是https://github.com/PacktPublishing/Python-Machine-Learning-Cookbook。另外，我们在https://github.com/PacktPublishing上还有其他书的代码包和视频，请需要的读者自行下载。

下载本书的彩色图片

我们也为你提供了一份PDF文件，里面包含了书中的截屏和图表等彩色图片，彩色图片能帮助你更好地理解输出的变化。下载网址为https://www.packtpub.com/sites/default/files/downloads/PythonMachineLearningCookbook_ColorImages.pdf。

勘误

虽然我们已尽力确保本书内容正确，但出错仍旧在所难免。如果你在书中发现错误，不管是文本还是代码，希望能告知我们，我们将不胜感激。这样做，你可以使其他读者免受挫败，也可以帮助我们改进本书的后续版本。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata，选择本书，单击**勘误表提交表单**（Errata Submission Form）的链接，并输入详细说明。[①]勘误一经核实，你提交的内容将被接受，此勘误会上传到本公司网站或添加到现有勘误表。

访问https://www.packtpub.com/books/content/support，在搜索框中输入书名，可以在**勘误**（Errata）部分查看已经提交的勘误信息。

盗版

任何媒体都会面临版权内容在互联网上的盗版问题，Packt也不例外。Packt非常重视版权保护。如果你发现我们的作品在互联网上被非法复制，不管以什么形式，都请立即为我们提供相关网址或网站名称，以便我们寻求补救。

请把可疑盗版材料的链接发到copyright@packtpub.com。

保护我们的作者，就是保护我们继续为你带来价值的能力，我们将不胜感激。

① 中文版勘误可以到http://www.ituring.com.cn/book/1894查看和提交。——编者注

问题

如果你对本书内容存有疑问，不管是哪个方面的，都可以通过questions@packtpub.com联系我们，我们会尽最大努力解决。

电子书

扫描如下二维码，即可获得本书电子版。

目　　录

第 1 章

监督学习

在这一章，我们将介绍以下主题：

- 数据预处理技术
- 标记编码方法
- 创建线性回归器（linear regressor）
- 计算回归准确性
- 保存模型数据
- 创建岭回归器（ridge regressor）
- 创建多项式回归器（polynomial regressor）
- 估算房屋价格
- 计算特征的相对重要性
- 评估共享单车的需求分布

1.1 简介

如果你熟悉机器学习的基础知识，那么肯定知道什么是监督学习。监督学习是指在有标记的样本（labeled samples）上建立机器学习的模型。例如，如果用尺寸、位置等不同参数建立一套模型来评估一栋房子的价格，那么首先需要创建一个数据库，然后为参数打上标记。我们需要告诉算法，什么样的参数（尺寸、位置）对应什么样的价格。有了这些带标记的数据，算法就可以学会如何根据输入的参数计算房价了。

无监督学习与刚才说的恰好相反，它面对的是没有标记的数据。假设需要把一些数据分成不同的组别，但是对分组的条件毫不知情，于是，无监督学习算法就会以最合理的方式将数据集分成确定数量的组别。我们将在后面章节介绍无监督学习。

建立书中的各种模型时，将使用许多Python程序包，像NumPy、SciPy、scikit-learn、matplotlib等。如果你使用Windows系统，推荐安装兼容SciPy关联程序包的Python发行版，网址为http://www.scipy.org/install.html，这些Python发行版里已经集成了常用的程序包。如果你使用

Mac OS X或者Ubuntu系统，安装这些程序包就相当简单了。下面列出来程序包安装和使用文档的链接：

- NumPy：http://docs.scipy.org/doc/numpy-1.10.1/user/install.html
- SciPy：http://www.scipy.org/install.html
- scikit-learn：http://scikit-learn.org/stable/install.html
- matplotlib：http://matplotlib.org/1.4.2/users/installing.html

现在，请确保你的计算机已经安装了所有程序包。

1.2　数据预处理技术

在真实世界中，经常需要处理大量的原始数据，这些原始数据是机器学习算法无法理解的。为了让机器学习算法理解原始数据，需要对数据进行预处理。

1.2.1　准备工作

来看看Python是如何对数据进行预处理的。首先，用你最喜欢的文本编辑器打开一个扩展名为.py的文件，例如preprocessor.py。然后在文件里加入下面两行代码：

```
import numpy as np
from sklearn import preprocessing
```

我们只是加入了两个必要的程序包。接下来创建一些样本数据。向文件中添加下面这行代码：

```
data = np.array([[3,  -1.5,    2,  -5.4], [0,    4,    -0.3,  2.1], [1,    3.3,
-1.9, -4.3]])
```

现在就可以对数据进行预处理了。

1.2.2　详细步骤

数据可以通过许多技术进行预处理，接下来将介绍一些最常用的预处理技术。

1. 均值移除（Mean removal）

通常我们会把每个特征的平均值移除，以保证特征均值为0（即标准化处理）。这样做可以消除特征彼此间的偏差（bias）。将下面几行代码加入之前打开的Python文件中：

```
data_standardized = preprocessing.scale(data)
print "\nMean =", data_standardized.mean(axis=0)
print "Std deviation =", data_standardized.std(axis=0)
```

现在来运行代码。打开命令行工具，然后输入以下命令：

```
$ python preprocessor.py
```

命令行工具中将显示以下结果：

```
Mean = [  5.55111512e-17  -1.11022302e-16  -7.40148683e-17  -7.40148683e-17]
Std deviation = [ 1.  1.  1.  1.]
```

你会发现特征均值几乎是0，而且标准差为1。

2. 范围缩放（Scaling）

数据点中每个特征的数值范围可能变化很大，因此，有时将特征的数值范围缩放到合理的大小是非常重要的。在Python文件中加入下面几行代码，然后运行程序：

```
data_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
data_scaled = data_scaler.fit_transform(data)
print "\nMin max scaled data =", data_scaled
```

范围缩放之后，所有数据点的特征数值都位于指定的数值范围内。输出结果如下所示：

```
Min max scaled data:
[[ 1.            0.            1.            0.          ]
 [ 0.            1.            0.41025641    1.          ]
 [ 0.33333333    0.87272727    0.            0.14666667]]
```

3. 归一化（Normalization）

数据归一化用于需要对特征向量的值进行调整时，以保证每个特征向量的值都缩放到相同的数值范围。机器学习中最常用的归一化形式就是将特征向量调整为L1范数，使特征向量的数值之和为1。增加下面两行代码到前面的Python文件中：

```
data_normalized = preprocessing.normalize(data, norm='l1')
print "\nL1 normalized data =", data_normalized
```

执行Python文件，就可以看到下面的结果：

```
L1    normalized    data:
[[    0.25210084    -0.12605042    0.16806723    -0.45378151]
 [    0.             0.625        -0.046875       0.328125  ]
 [    0.0952381      0.31428571   -0.18095238    -0.40952381]]
```

这个方法经常用于确保数据点没有因为特征的基本性质而产生较大差异，即确保数据处于同一数量级，提高不同特征数据的可比性。

4. 二值化（Binarization）

二值化用于将数值特征向量转换为布尔类型向量。增加下面两行代码到前面的Python文件中：

```
data_binarized = preprocessing.Binarizer(threshold=1.4).transform(data)
print "\nBinarized data =", data_binarized
```

再次执行Python文件，就可以看到下面的结果：

```
Binarized data:
[[    1.    0.    1.    0.]
 [    0.    1.    0.    1.]
 [    0.    1.    0.    0.]]
```

如果事先已经对数据有了一定的了解，就会发现使用这个技术的好处了。

5. 独热编码

通常，需要处理的数值都是稀疏地、散乱地分布在空间中，然而，我们并不需要存储这些大数值，这时就需要使用独热编码（One-Hot Encoding）。可以把独热编码看作是一种**收紧**（tighten）特征向量的工具。它把特征向量的每个特征与特征的非重复总数相对应，通过*one-of-k*的形式对每个值进行编码。特征向量的每个特征值都按照这种方式编码，这样可以更加有效地表示空间。例如，我们需要处理4维向量空间，当给一个特性向量的第*n*个特征进行编码时，编码器会遍历每个特征向量的第*n*个特征，然后进行非重复计数。如果非重复计数的值是*K*，那么就把这个特征转换为只有一个值是1其他值都是0的*K*维向量。增加下面几行代码到前面的Python文件中：

```
encoder = preprocessing.OneHotEncoder()
encoder.fit([[0, 2, 1, 12], [1, 3, 5, 3], [2, 3, 2, 12], [1, 2, 4, 3]])
encoded_vector = encoder.transform([[2, 3, 5, 3]]).toarray()
print "\nEncoded vector =", encoded_vector
```

结果如下所示：

```
Encoded vector:
[[ 0.  0.  1.  0.  1.  0.  0.  0.  1.  1.  0.]]
```

在上面的示例中，观察一下每个特征向量的第三个特征，分别是1、5、2、4这4个不重复的值，也就是说独热编码向量的长度是4。如果你需要对5进行编码，那么向量就是[0, 1, 0, 0]。向量中只有一个值是1。第二个元素是1，对应的值是5。

1.3　标记编码方法

在监督学习中，经常需要处理各种各样的标记。这些标记可能是数字，也可能是单词。如果标记是数字，那么算法可以直接使用它们，但是，许多情况下，标记都需要以人们可理解的形式存在，因此，人们通常会用单词标记训练数据集。标记编码就是要把单词标记转换成数值形式，让算法懂得如何操作标记。接下来看看如何标记编码。

详细步骤

(1) 新建一个Python文件，然后导入preprocessing程序包：

```
from sklearn import preprocessing
```

(2) 这个程序包包含许多数据预处理需要的函数。定义一个标记编码器（label encoder），代码如下所示：

```
label_encoder = preprocessing.LabelEncoder()
```

(3) `label_encoder`对象知道如何理解单词标记。接下来创建一些标记：

```
input_classes = ['audi', 'ford', 'audi', 'toyota', 'ford', 'bmw']
```

(4) 现在就可以为这些标记编码了：

```
label_encoder.fit(input_classes)
print "\nClass mapping:"
for i, item in enumerate(label_encoder.classes_):
    print item, '-->', i
```

(5) 运行代码，命令行工具中显示下面的结果：

```
Class mapping:
audi --> 0
bmw --> 1
ford --> 2
toyota --> 3
```

(6) 就像前面结果显示的那样，单词被转换成从0开始的索引值。现在，如果你遇到一组标记，就可以非常轻松地转换它们了，如下所示：

```
labels = ['toyota', 'ford', 'audi']
encoded_labels = label_encoder.transform(labels)
print "\nLabels =", labels
print "Encoded labels =", list(encoded_labels)
```

命令行工具中将显示下面的结果：

```
Labels = ['toyota', 'ford', 'audi']
Encoded labels = [3, 2, 0]
```

(7) 这种方式比纯手工进行单词与数字的编码要简单许多。还可以通过数字反转回单词的功能检查结果的正确性：

```
encoded_labels = [2, 1, 0, 3, 1]
decoded_labels = label_encoder.inverse_transform(encoded_labels)
print "\nEncoded labels =", encoded_labels
print "Decoded labels =", list(decoded_labels)
```

结果如下所示：

```
Encoded labels = [2, 1, 0, 3, 1]
Decoded labels = ['ford', 'bmw', 'audi', 'toyota', 'bmw']
```

可以看到，映射结果是完全正确的。

1.4　创建线性回归器

回归是估计输入数据与连续值输出数据之间关系的过程。数据通常是实数形式的，我们的目标是估计满足输入到输出映射关系的基本函数。让我们从一个简单的示例开始。考虑下面的输入与输出映射关系：

$1 \rightarrow 2$

$3 \rightarrow 6$

$4.3 \rightarrow 8.6$

$7.1 \rightarrow 14.2$

如果要你估计输入与输出的关联关系，你可以通过模式匹配轻松地找到结果。我们发现输出结果一直是输入数据的两倍，因此输入与输出的转换公式就是这样：

$$f(x) = 2x$$

这是体现输入值与输出值关联关系的一个简单函数。但是，在真实世界中通常都不会这么简单，输入与输出的映射关系函数并不是一眼就可以看出来的。

1.4.1　准备工作

线性回归用输入变量的线性组合来估计基本函数。前面的示例就是一种单输入单输出变量的线性回归。

现在考虑如图1-1所示的情况。

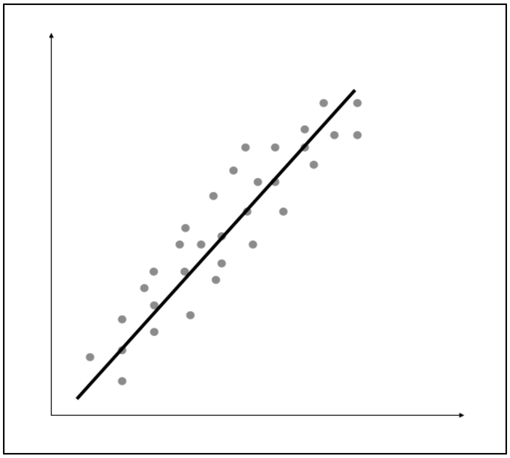


图　1-1

线性回归的目标是提取输入变量与输出变量的关联线性模型，这就要求实际输出与线性方程预测的输出的残差平方和（sum of squares of differences）最小化。这种方法被称为**普通最小二乘法**（Ordinary Least Squares，OLS）。

你可能觉得用一条曲线对这些点进行拟合效果会更好，但是线性回归不允许这样做。线性回归的主要优点就是方程简单。如果你想用非线性回归，可能会得到更准确的模型，但是拟合速度会慢很多。线性回归模型就像前面那张图里显示的，用一条直线近似数据点的趋势。接下来看看如何用Python建立线性回归模型。

1.4.2 详细步骤

假设你已经创建了数据文件data_singlevar.txt，文件里用逗号分隔符分割字段，第一个字段是输入值，第二个字段是与逗号前面的输入值相对应的输出值。你可以用这个文件作为输入参数。

(1) 创建一个Python文件regressor.py，然后在里面增加下面几行代码：

```
import sys
import numpy as np
filename = sys.argv[1]
X = []
y = []
with open(filename, 'r') as f:
    for line in f.readlines():
        xt, yt = [float(i) for i in line.split(',')]
        X.append(xt)
        y.append(yt)
```

把输入数据加载到变量`X`和`y`，其中`X`是数据，`y`是标记。在代码的`for`循环体中，我们解析每行数据，用逗号分割字段。然后，把字段转化为浮点数，并分别保存到变量`X`和`y`中。

(2) 建立机器学习模型时，需要用一种方法来验证模型，检查模型是否达到一定的满意度（satisfactory level）。为了实现这个方法，把数据分成两组：训练数据集（training dataset）与测试数据集（testing dataset）。训练数据集用来建立模型，测试数据集用来验证模型对未知数据的学习效果。因此，先把数据分成训练数据集与测试数据集：

```
num_training = int(0.8 * len(X))
num_test = len(X) - num_training

# 训练数据
X_train = np.array(X[:num_training]).reshape((num_training,1))
y_train = np.array(y[:num_training])

# 测试数据
X_test = np.array(X[num_training:]).reshape((num_test,1))
y_test = np.array(y[num_training:])
```

这里用80%的数据作为训练数据集，其余20%的数据作为测试数据集。

(3) 现在已经准备好训练模型。接下来创建一个回归器对象，代码如下所示：

```
from sklearn import linear_model

# 创建线性回归对象
linear_regressor = linear_model.LinearRegression()

# 用训练数据集训练模型
linear_regressor.fit(X_train, y_train)
```

(4) 我们利用训练数据集训练了线性回归器。向`fit`方法提供输入数据即可训练模型。用下面的代码看看它如何拟合：

```
import matplotlib.pyplot as plt

y_train_pred = linear_regressor.predict(X_train)
plt.figure()
plt.scatter(X_train, y_train, color='green')
plt.plot(X_train, y_train_pred, color='black', linewidth=4)
plt.title('Training data')
plt.show()
```

(5) 在命令行工具中执行如下命令：

```
$ python regressor.py data_singlevar.txt
```

就会看到如图1-2所示的线性回归。

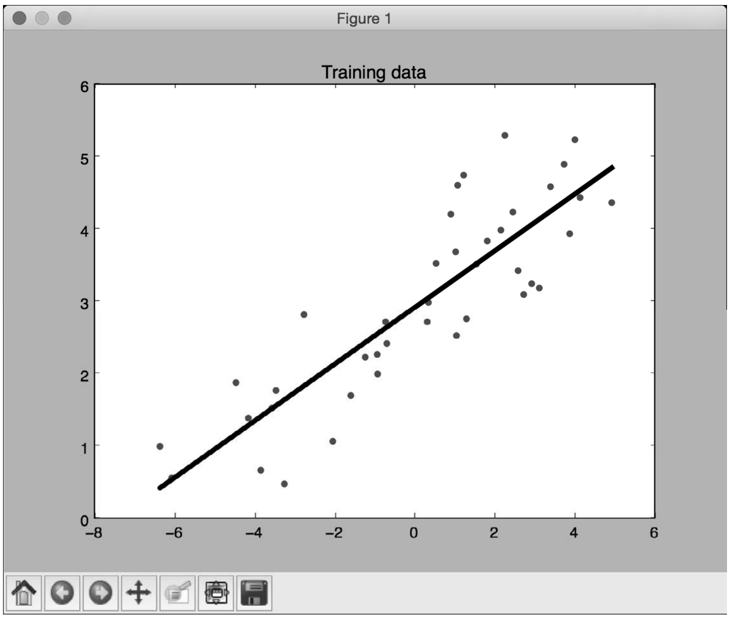


图　1-2

(6) 在前面的代码中，我们用训练的模型预测了训练数据的输出结果，但这并不能说明模型对未知的数据也适用，因为我们只是在训练数据上运行模型。这只能体现模型对训练数据的拟合效果。从图1-2中可以看到，模型训练的效果很好。

(7) 接下来用模型对测试数据集进行预测，然后画出来看看，代码如下所示：

```
y_test_pred = linear_regressor.predict(X_test)

plt.scatter(X_test, y_test, color='green')
plt.plot(X_test, y_test_pred, color='black', linewidth=4)
plt.title('Test data')
plt.show()
```

运行代码，可以看到如图1-3所示的线性回归。

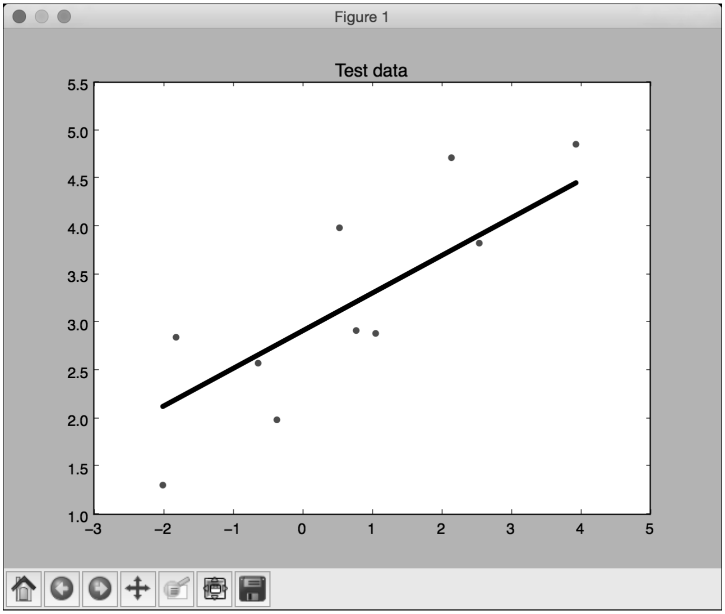


图　1-3

1.5　计算回归准确性

现在已经建立了回归器，接下来最重要的就是如何评价回归器的拟合效果。在模型评价的相关内容中，用**误差**（error）表示实际值与模型预测值之间的差值。

1.5.1　准备工作

下面快速了解几个衡量回归器拟合效果的重要指标（metric）。回归器可以用许多不同的指标

进行衡量，部分指标如下所示。

- 平均绝对误差（mean absolute error）：这是给定数据集的所有数据点的绝对误差平均值。
- 均方误差（mean squared error）：这是给定数据集的所有数据点的误差的平方的平均值。这是最流行的指标之一。
- 中位数绝对误差（median absolute error）：这是给定数据集的所有数据点的误差的中位数。这个指标的主要优点是可以消除异常值（outlier）的干扰。测试数据集中的单个坏点不会影响整个误差指标，均值误差指标会受到异常点的影响。
- 解释方差分（explained variance score）：这个分数用于衡量我们的模型对数据集波动的解释能力。如果得分1.0分，那么表明我们的模型是完美的。
- R方得分（R2 score）：这个指标读作"R方"，是指确定性相关系数，用于衡量模型对未知样本预测的效果。最好的得分是1.0，值也可以是负数。

1.5.2　详细步骤

scikit-learn里面有一个模块，提供了计算所有指标的功能。重新打开一个Python文件，然后输入以下代码：

```
import sklearn.metrics as sm

print "Mean absolute error =", round(sm.mean_absolute_error(y_test, y_test_pred), 2)
print "Mean squared error =", round(sm.mean_squared_error(y_test, y_ test_pred), 2)
print "Median absolute error =", round(sm.median_absolute_error(y_ test, y_test_pred),
2)
print "Explained variance score =", round(sm.explained_variance_ score(y_test,
y_test_pred), 2)
print "R2 score =", round(sm.r2_score(y_test, y_test_pred), 2)
```

每个指标都描述得面面俱到是非常乏味的，因此只选择一两个指标来评估我们的模型。通常的做法是尽量保证均方误差最低，而且解释方差分最高。

1.6　保存模型数据

模型训练结束之后，如果能够把模型保存成文件，那么下次再使用的时候，只要简单地加载就可以了。

详细步骤

用程序保存模型的具体操作步骤如下。

(1) 在Python文件regressor.py中加入以下代码：

```
import cPickle as pickle

output_model_file = 'saved_model.pkl'
with open(output_model_file, 'w') as f:
    pickle.dump(linear_regressor, f)
```

(2) 回归模型会保存在saved_model.pkl文件中。下面看看如何加载并使用它，代码如下所示：

```
with open(output_model_file, 'r') as f:
    model_linregr = pickle.load(f)

y_test_pred_new = model_linregr.predict(X_test)
print "\nNew mean absolute error =", round(sm.mean_absolute_ error(y_test,
y_test_pred_new), 2)
```

(3) 这里只是把回归模型从Pickle文件加载到`model_linregr`变量中。你可以将打印结果与前面的结果进行对比，确认模型与之前的一样。

1.7 创建岭回归器

线性回归的主要问题是对异常值敏感。在真实世界的数据收集过程中，经常会遇到错误的度量结果。而线性回归使用的普通最小二乘法，其目标是使平方误差最小化。这时，由于异常值误差的绝对值很大，因此会引起问题，从而破坏整个模型。

1.7.1 准备工作

先看图1-4。

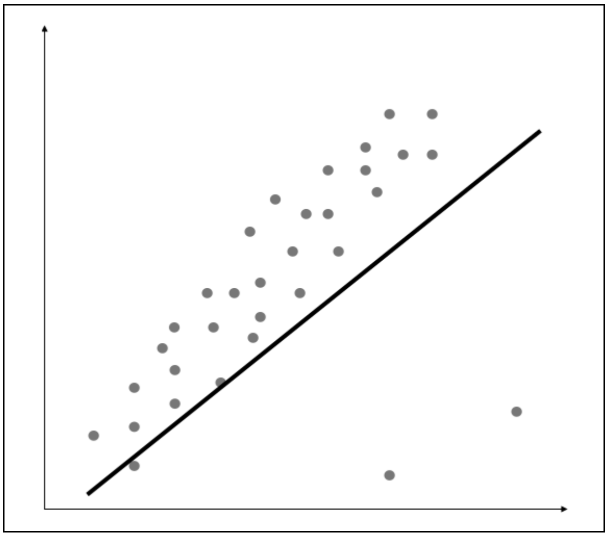


图 1-4

右下角的两个数据点明显是异常值，但是这个模型需要拟合所有的数据点，因此导致整个模型都错了。仅凭直觉观察，我们就会觉得如图1-5的拟合结果更好。

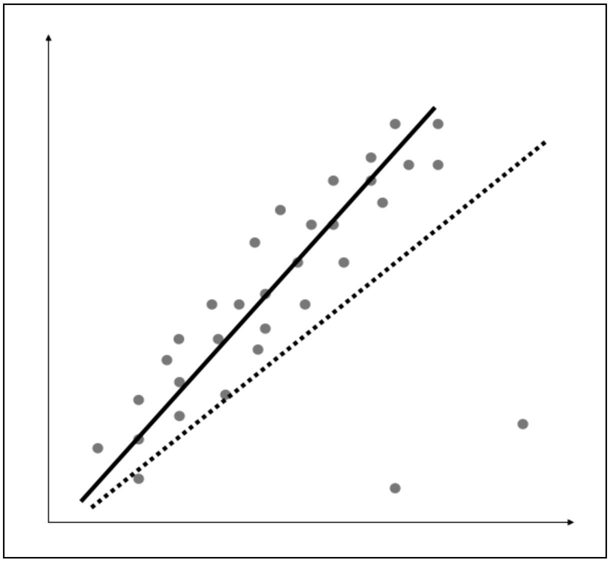


图　1-5

普通最小二乘法在建模时会考虑每个数据点的影响，因此，最终模型就会像图1-4显示的直线那样。显然，我们发现这个模型不是最优的。为了避免这个问题，我们引入**正则化项**的系数作为阈值来消除异常值的影响。这个方法被称为**岭回归**。

1.7.2　详细步骤

接下来看看如何用Python建立岭回归器。

(1) 你可以从data_multi_variable.txt文件中加载数据。这个文件的每一行都包含多个数值。除了最后一个数值外，前面的所有数值构成输入特征向量。

(2) 把下面的代码加入regressor.py文件中。我们用一些参数初始化岭回归器：

```
ridge_regressor = linear_model.Ridge(alpha=0.01, fit_ intercept=True,
max_iter=10000)
```

(3) `alpha`参数控制回归器的复杂程度。当`alpha`趋于`0`时，岭回归器就是用普通最小二乘法的线性回归器。因此，如果你希望模型对异常值不那么敏感，就需要设置一个较大的`alpha`值。这里把`alpha`值设置为`0.01`。

(4) 下面让我们来训练岭回归器。

```
ridge_regressor.fit(X_train, y_train)
y_test_pred_ridge = ridge_regressor.predict(X_test)
print "Mean absolute error =", round(sm.mean_absolute_error
    (y_ test, y_test_pred_ridge), 2)
print "Mean squared error =", round(sm.mean_squared_error
    (y_test, y_test_pred_ridge), 2)
print "Median absolute error =", round(sm.median_absolute_error
    (y_ test, y_test_pred_ridge), 2)
print "Explain variance score =", round(sm.explained_variance_ score
    (y_test, y_test_pred_ridge), 2)
print "R2 score =", round(sm.r2_score(y_test, y_test_pred_ridge), 2)
```

运行代码检查误差指标。可以用同样的数据建立一个线性回归器，并与岭回归器的结果进行比较，看看把正则化引入回归模型之后的效果如何。

1.8 创建多项式回归器

线性回归模型有一个主要的局限性，那就是它只能把输入数据拟合成直线，而多项式回归模型通过拟合多项式方程来克服这类问题，从而提高模型的准确性。

1.8.1 准备工作

先看图1-6。

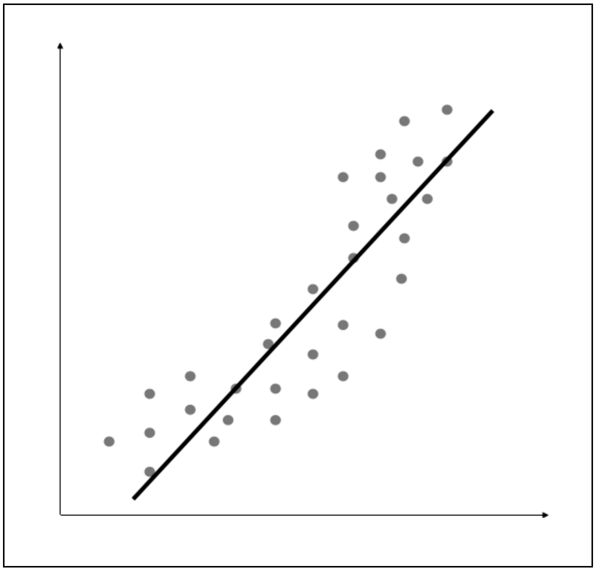


图 1-6

从图1-6中可以看到，数据点本身的模式中带有自然的曲线，而线性模型是不能捕捉到这一点的。再来看看多项式模型的效果，如图1-7所示。

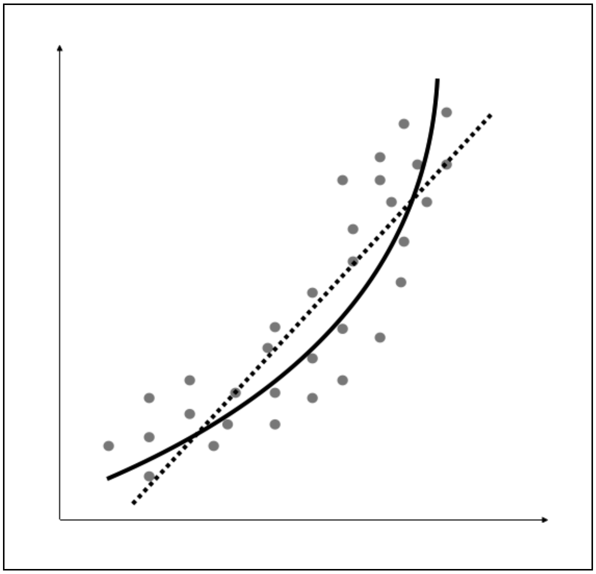


图　1-7

图1-7中的虚线表示线性回归模型，实线表示多项式回归模型。这个模型的曲率是由多项式的次数决定的。随着模型曲率的增加，模型变得更准确。但是，增加曲率的同时也增加了模型的复杂性，因此拟合速度会变慢。当我们对模型的准确性的理想追求与计算能力限制的残酷现实发生冲突时，就需要综合考虑了。

1.8.2　详细步骤

(1) 将下面的代码加入Python文件regressor.py中：

```
from sklearn.preprocessing import PolynomialFeatures

polynomial = PolynomialFeatures(degree=3)
```

(2) 上一行将曲线的多项式的次数的初始值设置为3。下面用数据点来计算多项式的参数：

```
X_train_transformed = polynomial.fit_transform(X_train)
```

其中，`X_train_transformed`表示多项式形式的输入，与线性回归模型是一样大的。

(3) 接下来用文件中的第一个数据点来检查多项式模型是否能够准确预测：

```
datapoint = [0.39,2.78,7.11]
poly_datapoint = polynomial.fit_transform(datapoint)

poly_linear_model = linear_model.LinearRegression()
poly_linear_model.fit(X_train_transformed, y_train)
print "\nLinear regression:", linear_regressor.predict(datapoint) [0]
print "\nPolynomial regression:", poly_linear_model.predict(poly_datapoint)[0]
```

多项式回归模型计算变量数据点的值恰好就是输入数据文件中的第一行数据值。再用线性回归模型测试一下，唯一的差别就是展示数据的形式。运行代码，可以看到下面的结果：

```
Linear regression: -11.0587294983
Polynomial regression: -10.9480782122
```

可以发现，多项式回归模型的预测值更接近实际的输出值。如果想要数据更接近实际输出值，就需要增加多项式的次数。

(4) 将多项式的次数加到10看看结果：

```
polynomial = PolynomialFeatures(degree=10)
```

可以看到下面的结果：

```
Polynomial regression: -8.20472183853
```

现在，你可以发现预测值与实际的输出值非常地接近。

1.9 估算房屋价格

是时候用所学的知识来解决真实世界的问题了。让我们用这些原理来估算房屋价格。房屋估价是理解回归分析最经典的案例之一，通常是一个不错的切入点。它符合人们的直觉，而且与人们的生活息息相关，因此在用机器学习处理复杂事情之前，通过房屋估价可以更轻松地理解相关概念。我们将使用带AdaBoost算法的**决策树回归器**（decision tree regressor）来解决这个问题。

1.9.1 准备工作

决策树是一个树状模型，每个节点都做出一个决策，从而影响最终结果。叶子节点表示输出数值，分支表示根据输入特征做出的中间决策。AdaBoost算法是指自适应增强（adaptive boosting）算法，这是一种利用其他系统增强模型准确性的技术。这种技术是将不同版本的算法结果进行组合，用加权汇总的方式获得最终结果，被称为**弱学习器**（weak learners）。AdaBoost算法在每个阶段获取的信息都会反馈到模型中，这样学习器就可以在后一阶段重点训练难以分类的样本。这种学习方式可以增强系统的准确性。

首先使用AdaBoost算法对数据集进行回归拟合，再计算误差，然后根据误差评估结果，用同样的数据集重新拟合。可以把这些看作是回归器的调优过程，直到达到预期的准确性。假设你拥

有一个包含影响房价的各种参数的数据集，我们的目标就是估计这些参数与房价的关系，这样就可以根据未知参数估计房价了。

1.9.2　详细步骤

(1) 创建一个新的Python文件housing.py，然后加入下面的代码：

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn import datasets
from sklearn.metrics import mean_squared_error, explained_variance_score
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
```

(2) 网上有一个标准房屋价格数据库，人们经常用它来研究机器学习。你可以在https://archive.ics.uci.edu/ml/datasets/Housing下载数据。不过scikit-learn提供了数据接口，可以直接通过下面的代码加载数据：

```
housing_data = datasets.load_boston()
```

每个数据点由影响房价的13个输入参数构成。你可以用`housing_data.data`获取输入的数据，用`housing_data.target`获取对应的房屋价格。

(3) 接下来把输入数据与输出结果分成不同的变量。我们可以通过`shuffle`函数把数据的顺序打乱：

```
X, y = shuffle(housing_data.data, housing_data.target, random_ state=7)
```

(4) 参数`random_state`用来控制如何打乱数据，让我们可以重新生成结果。接下来把数据分成训练数据集和测试数据集，其中80%的数据用于训练，剩余20%的数据用于测试：

```
num_training = int(0.8 * len(X))
X_train, y_train = X[:num_training], y[:num_training]
X_test, y_test = X[num_training:], y[num_training:]
```

(5) 现在已经可以拟合一个决策树回归模型了。选一个最大深度为4的决策树，这样可以限制决策树不变成任意深度：

```
dt_regressor = DecisionTreeRegressor(max_depth=4)
dt_regressor.fit(X_train, y_train)
```

(6) 再用带AdaBoost算法的决策树回归模型进行拟合：

```
ab_regressor =AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
n_estimators=400, random_state=7)
ab_regressor.fit(X_train, y_train)
```

这样可以帮助我们对比训练效果，看看AdaBoost算法对决策树回归器的训练效果有多大改善。

(7) 接下来评价决策树回归器的训练效果：

```
y_pred_dt = dt_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred_dt)
evs = explained_variance_score(y_test, y_pred_dt)
print "\n#### Decision Tree performance ####"
print "Mean squared error =", round(mse, 2)
print "Explained variance score =", round(evs, 2)
```

(8) 现在评价一下AdaBoost算法改善的效果：

```
y_pred_ab = ab_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred_ab)
evs = explained_variance_score(y_test, y_pred_ab)
print "\n#### AdaBoost performance ####"
print "Mean squared error =", round(mse, 2)
print "Explained variance score =", round(evs, 2)
```

(9) 命令行工具显示的输出结果如下所示：

```
#### 决策树学习效果 ####
Mean squared error = 14.79
Explained variance score = 0.82

#### AdaBoost算法改善效果 ####
Mean squared error = 7.54
Explained variance score = 0.91
```

前面的结果表明，AdaBoost算法可以让误差更小，且解释方差分更接近1。

1.10 计算特征的相对重要性

所有特征都同等重要吗？在这个案例中，我们用了13个特征，它们对模型都有贡献。但是，有一个重要的问题出现了：如何判断哪个特征更加重要？显然，所有的特征对结果的贡献是不一样的。如果需要忽略一些特征，就需要知道哪些特征不太重要。scikit-learn里面有这样的功能。

详细步骤

(1) 画出特征的相对重要性，在housing.py文件中加入下面几行代码：

```
plot_feature_importances(dt_regressor.feature_importances_,
        'Decision Tree regressor', housing_data.feature_names)
plot_feature_importances(ab_regressor.feature_importances_,
        'AdaBoost regressor', housing_data.feature_names)
```

回归器对象有一个`feature_importances_`方法会告诉我们每个特征的相对重要性。

(2) 接下来需要定义`plot_feature_importances`来画出条形图：

```
def plot_feature_importances(feature_importances, title, feature_names):
    # 将重要性值标准化
    feature_importances = 100.0 * (feature_importances / max(feature_importances))

    # 将得分从高到低排序
    index_sorted = np.flipud(np.argsort(feature_importances))

    # 让X坐标轴上的标签居中显示
    pos = np.arange(index_sorted.shape[0]) + 0.5

    # 画条形图
    plt.figure()
    plt.bar(pos, feature_importances[index_sorted], align='center')
    plt.xticks(pos, feature_names[index_sorted])
    plt.ylabel('Relative Importance')
    plt.title(title)
    plt.show()
```

(3) 我们从`feature_importances_`方法里取值，然后把数值放大到0~100的范围内。运行前面的代码，可以看到两张图（不带AdaBoost算法与带AdaBoost算法两种模型）。仔细观察图1-8和图1-9，看看能从决策树回归器中获得什么。

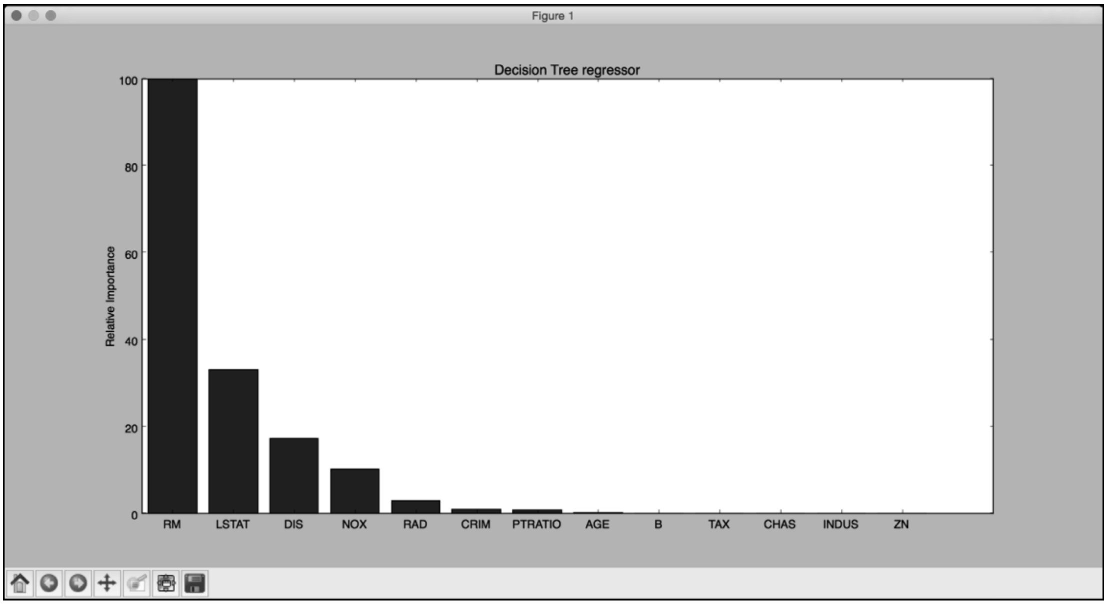


图　1-8

(4) 从图1-8可以发现，不带AdaBoost算法的决策树回归器显示的最重要特征是RM。再看看带AdaBoost算法的决策树回归器的特征重要性排序条形图，如图1-9所示。

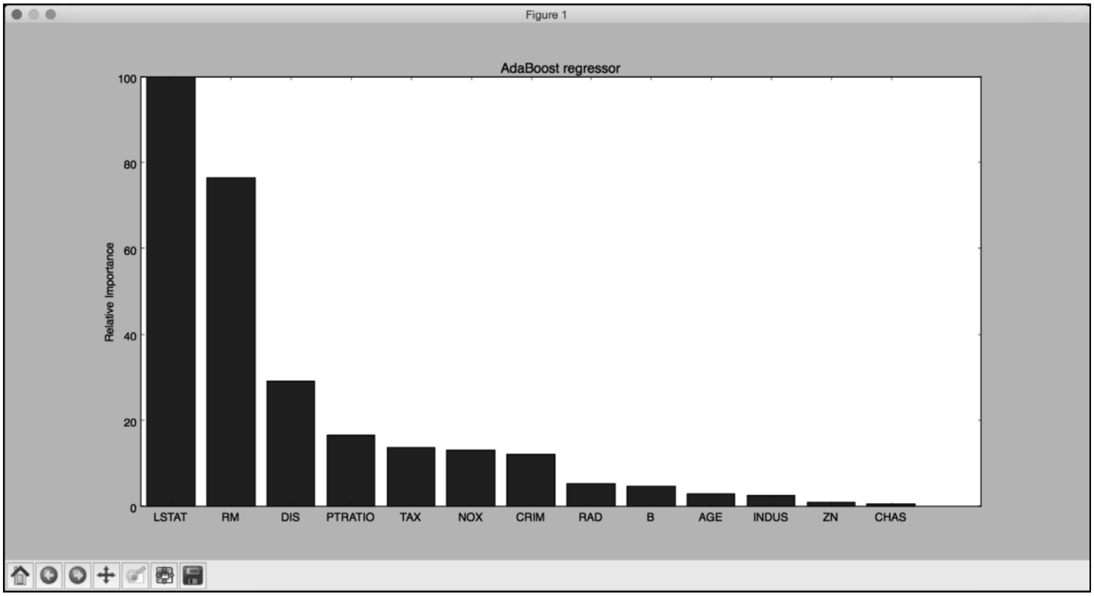


图 1-9

加入AdaBoost算法之后，房屋估价模型的最重要特征是LSTAT。在现实生活中，如果对这个数据集建立不同的回归器，就会发现最重要的特征是LSTAT，这足以体现AdaBoost算法对决策树回归器训练效果的改善。

1.11 评估共享单车的需求分布

本节将用一种新的回归方法解决共享单车的需求分布问题。我们采用**随机森林回归器**（random forest regressor）估计输出结果。随机森林是一个决策树集合，它基本上就是用一组由数据集的若干子集构建的决策树构成，再用决策树平均值改善整体学习效果。

1.11.1 准备工作

我们将使用bike_day.csv文件中的数据集，它可以在 https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset 获取。这份数据集一共16列，前两列是序列号与日期，分析的时候可以不用；最后三列数据是不同类型的输出结果；最后一列是第十四列与第十五列的和，因此建立模型时可以不考虑第十四列与第十五列。

1.11.2 详细步骤

接下来看看Python如何解决这个问题。如果你下载了本书源代码，就可以看到bike_ sharing.py

文件里已经包含了完整代码。这里将介绍若干重要的部分。

(1) 首先导入一些新的程序包，如下：

```
import csv
from sklearn.ensemble import RandomForestRegressor
from housing import plot_feature_importances
```

(2) 我们需要处理CSV文件，因此加入了csv程序包来读取CSV文件。由于这是一个全新的数据集，因此需要自己定义一个数据集加载函数：

```
def load_dataset(filename):
    file_reader = csv.reader(open(filename, 'rb'), delimiter=',')
    X, y = [], []
    for row in file_reader:
        X.append(row[2:13])
        y.append(row[-1])

    # 提取特征名称
    feature_names = np.array(X[0])

    # 将第一行特征名称移除，仅保留数值
    return np.array(X[1:]).astype(np.float32), np.array(y[1:]).astype(np.float32),
feature_names
```

在这个函数中，我们从CSV文件读取了所有数据。把数据显示在图形中时，特征名称非常有用。把特征名称数据从输入数值中分离出来，并作为函数返回值。

(3) 读取数据，并打乱数据顺序，让新数据与原来文件中数据排列的顺序没有关联性：

```
X, y, feature_names = load_dataset(sys.argv[1])
X, y = shuffle(X, y, random_state=7)
```

(4) 和之前的做法一样，需要将数据分成训练数据和测试数据。这一次，我们将90%的数据用于训练，剩余10%的数据用于测试：

```
num_training = int(0.9 * len(X))
X_train, y_train = X[:num_training], y[:num_training]
X_test, y_test = X[num_training:], y[num_training:]
```

(5) 下面开始训练回归器：

```
rf_regressor = RandomForestRegressor(n_estimators=1000, max_depth=10,
min_samples_split=1)
rf_regressor.fit(X_train, y_train)
```

其中，参数`n_estimators`是指评估器（estimator）的数量，表示随机森林需要使用的决策树数量；参数`max_depth`是指每个决策树的最大深度；参数`min_samples_split`是指决策树分裂一个节点需要用到的最小数据样本量。

(6) 评价随机森林回归器的训练效果：

```
y_pred = rf_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
evs = explained_variance_score(y_test, y_pred)
print "\n#### Random Forest regressor performance ####"
print "Mean squared error =", round(mse, 2)
print "Explained variance score =", round(evs, 2)
```

(7) 由于已经有画出特征重要性条形图的函数`plot_feature_importances`了，接下来直接调用它：

```
plot_feature_importances(rf_regressor.feature_importances_, 'Random Forest
regressor', feature_names)
```

执行代码，可以看到如图1-10所示的图形。

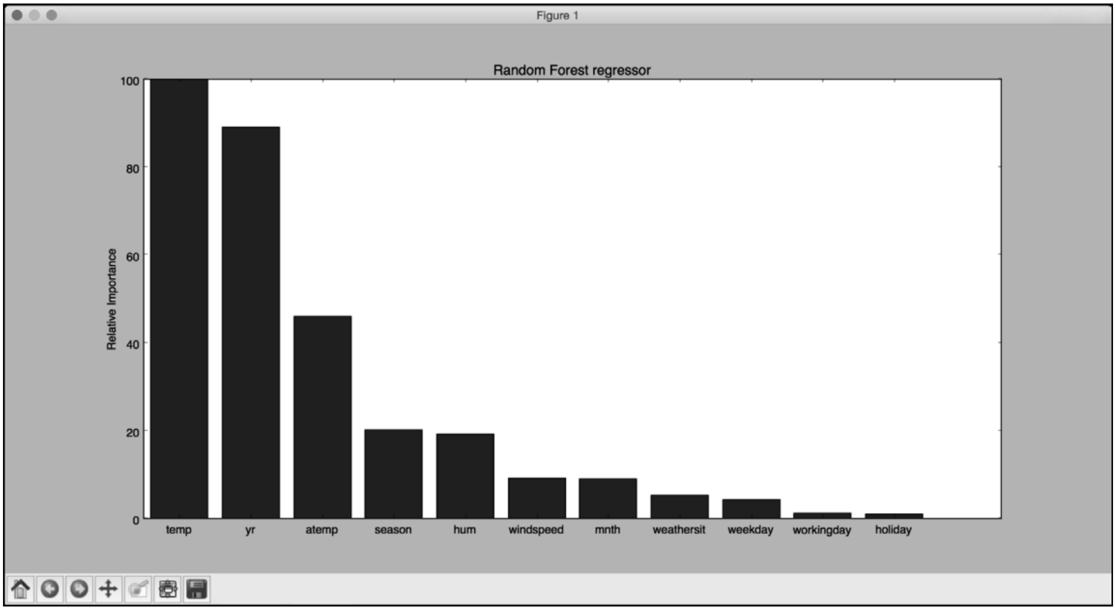


图 1-10

看来温度（temp）是影响自行车租赁的最重要因素。

1.11.3 更多内容

把第十四列与第十五列数据加入数据集，看看结果有什么区别。在新的特征重要性条形图中，除了这两个特征外，其他特征都变成了0。这是由于输出结果可以通过简单地对第十四列与第十五列数据求和得出，因此算法不需要其他特征计算结果。在数据集加载函数`load_dataset`中，

我们需要对for循环内的取值范围稍作调整：

```
X.append(row[2:15])
```

现在再画出特征重要性条形图，可以看到如图1-11所示的柱形图。

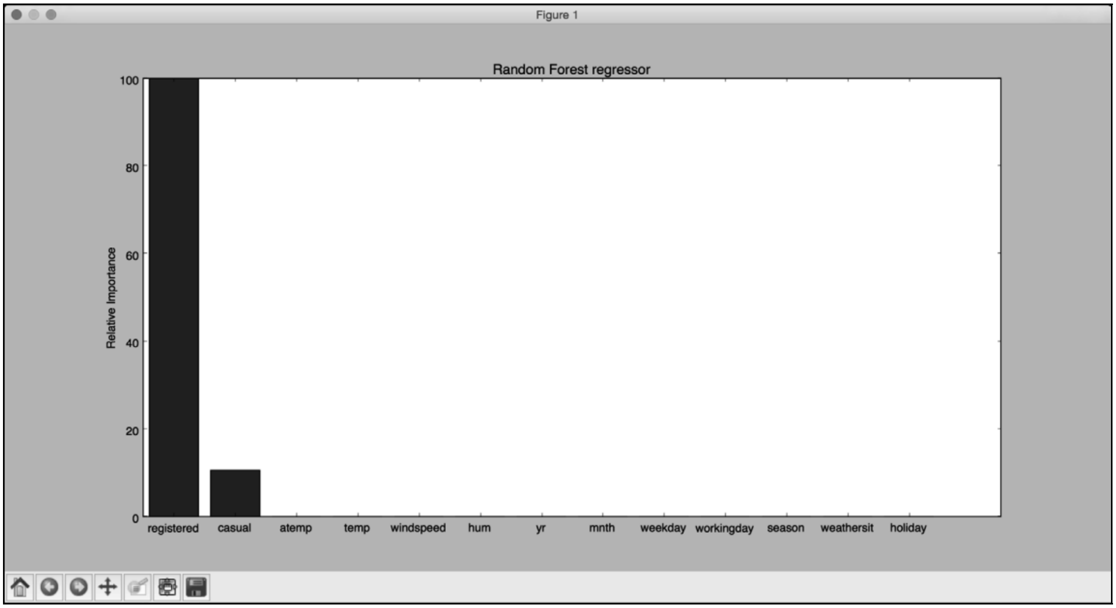


图　1-11

与预想的一样，从图中可以看出，只有这两个特征是重要的，这确实也符合常理，因为最终结果仅仅是这两个特征相加得到的。因此，这两个变量与输出结果有直接的关系，回归器也就认为它不需要其他特征来预测结果了。在消除数据集冗余变量方面，这是非常有用的工具。

还有一份按小时统计的自行车共享数据bike_hour.csv。我们需要用到第3~14列，因此先对数据集加载函数load_dataset做一点调整：

```
X.append(row[2:14])
```

运行代码，可以看到回归器的训练结果如下：

```
#### 随机森林学习效果 ####
Mean squared error = 2619.87
Explained variance score = 0.92
```

特征重要性条形图如图1-12所示。

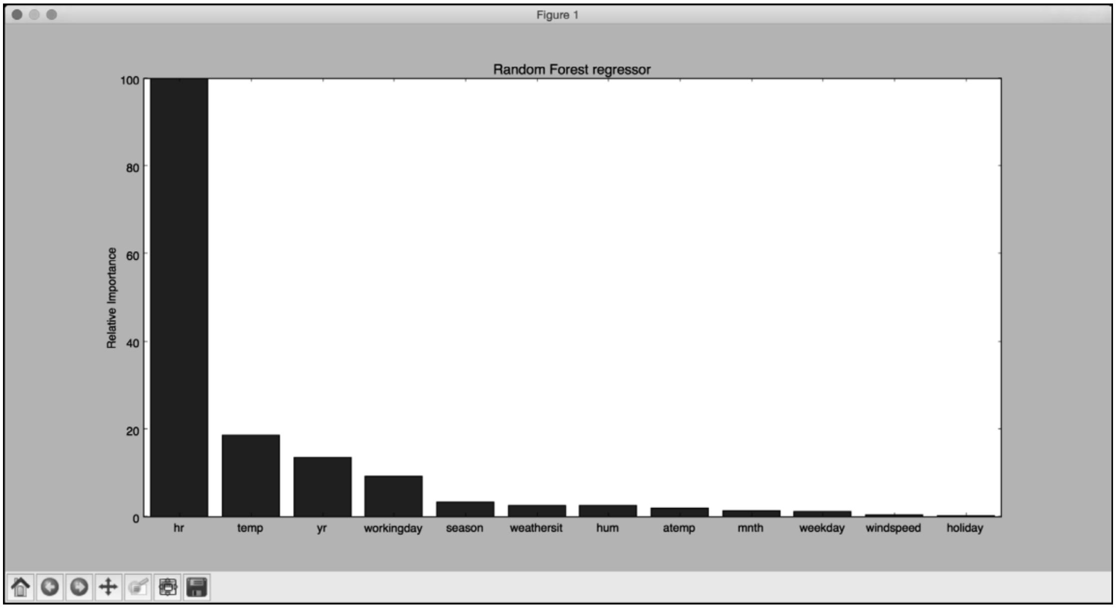


图　1-12

图1-12中显示，最重要的特征是一天中的不同时点（hr），这也完全符合人们的直觉；其次重要的是温度，与我们之前分析的结果一致。

第 2 章

创建分类器

2

在这一章，我们将介绍以下主题：

- 建立简单分类器（simple classifier）
- 建立逻辑回归分类器（logistic regression classifier）
- 建立朴素贝叶斯分类器（Naïve Bayes classifier）
- 将数据集分割成训练集和测试集
- 用交叉验证（cross-validation）检验模型准确性
- 混淆矩阵（confusion matrix）可视化
- 提取性能报告
- 根据汽车特征评估质量
- 生成验证曲线（validation curves）
- 生成学习曲线（learning curves）
- 估算收入阶层（income bracket）

2.1　简介

在机器学习领域中，分类是指利用数据的特性将其分成若干类型的过程。分类与上一章介绍的回归不同，回归的输出结果是实数。监督学习分类器就是用带标记的训练数据建立一个模型，然后对未知数据进行分类。

分类器可以是实现分类功能的任意算法，最简单的分类器就是简单的数学函数。在真实世界中，分类器可以是非常复杂的形式。在学习过程中，可以看到二元（binary）分类器，将数据分成两类，也可以看到多元（multiclass）分类器，将数据分成两个以上的类型。解决分类问题的数据手段都倾向于解决二元分类问题，可以通过不同的形式对其进行扩展，进而解决多元分类问题。

分类器准确性的估计是机器学习领域的重要内容。我们需要学会如何使用现有的数据获取新的思路（机器学习模型），然后把模型应用到真实世界中。在这一章里，我们将看到许多类似的主题。

2.2 建立简单分类器

本节学习如何用训练数据建立一个简单分类器。

2.2.1 详细步骤

(1) 使用simple_classifier.py文件作为参考。假设你已经和上一章一样导入了numpy和matplotlib.pyplot程序包，那么需要创建一些样本数据：

```
X = np.array([[3,1], [2,5], [1,8], [6,4], [5,2], [3,5], [4,7], [4,-1]])
```

(2) 为这些数据点分配一些标记：

```
y = [0, 1, 1, 0, 0, 1, 1, 0]
```

(3) 因为只有两个类，所以y列表包含0和1。一般情况下，如果你有N个类，那么y的取值范围就是从0到N-1。接下来按照类型标记把样本数据分成两类：

```
class_0 = np.array([X[i] for i in range(len(X)) if y[i]==0])
class_1 = np.array([X[i] for i in range(len(X)) if y[i]==1])
```

(4) 为了对数据有个直观的认识，把图像画出来，如下所示：

```
plt.figure()
plt.scatter(class_0[:,0], class_0[:,1], color='black', marker='s')
plt.scatter(class_1[:,0], class_1[:,1], color='black', marker='x')
```

这是一个散点图（scatterplot），用方块和叉表示两类数据。在前面的代码中，参数marker用来表示数据点的形状。用方块表示class_0的数据，用叉表示class_1的数据。运行代码，可以看到如图2-1所示的图形。

(5) 在之前的两行代码中，只是用变量X与y之间的映射关系创建了两个列表。如果要你直观地展示数据点的不同类型，在两类数据间画一条分割线，那么怎么实现呢？你只要用直线方程在两类数据之间画一条直线就可以了。下面看看如何实现：

```
line_x = range(10)
line_y = line_x
```

(6) 用数学公式$y = x$创建一条直线。代码如下所示：

```
plt.figure()
plt.scatter(class_0[:,0], class_0[:,1], color='black', marker='s')
plt.scatter(class_1[:,0], class_1[:,1], color='black', marker='x')
plt.plot(line_x, line_y, color='black', linewidth=3)
plt.show()
```

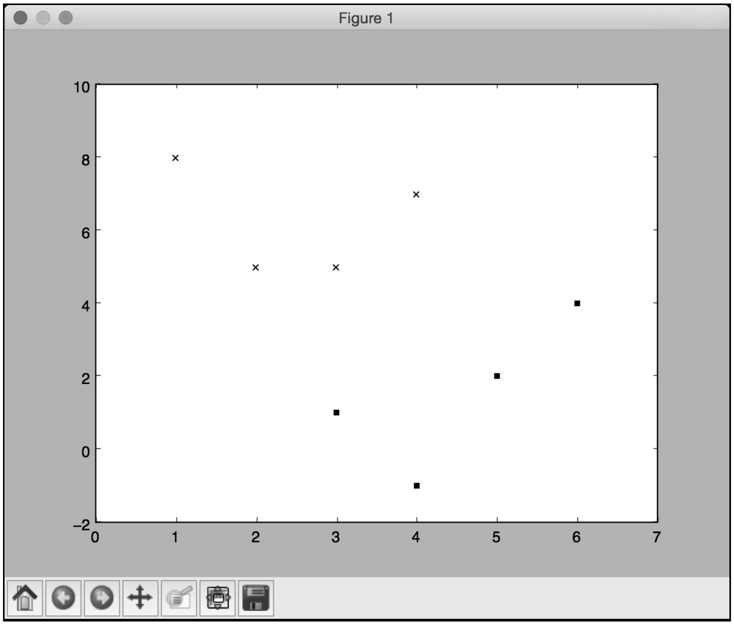


图　2-1

(7) 运行代码，可以看到如图2-2所示的图形。

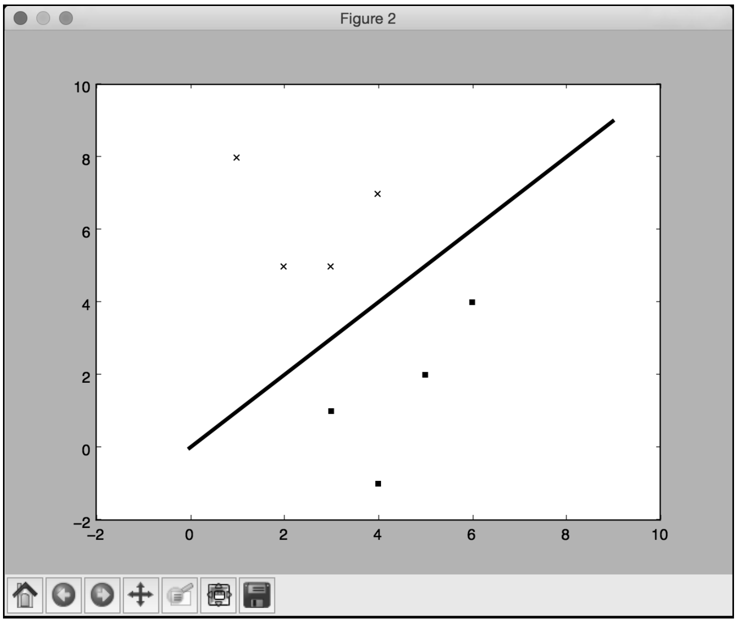


图　2-2

2.2.2　更多内容

用以下规则建立了一个简单的分类器：如果输入点(a, b)的a大于或等于b，那么它属于类型class_0；反之，它属于class_1。如果对数据点逐个进行检查，你会发现每个数都是这样，这样你就建立了一个可以识别未知数据的线性分类器（linear classifier）。之所以称其为线性分类器，是因为分割线是一条直线。如果分割线是一条曲线，就是非线性分类器（nonlinear classifier）。

这样简单的分类器之所以可行，是因为数据点很少，可以直观地判断分割线。如果有几千个数据点呢？如何对分类过程进行一般化处理（generalize）呢？下一节将介绍这一主题。

2.3　建立逻辑回归分类器

虽然这里也出现了上一章介绍的**回归**这个词，但逻辑回归其实是一种分类方法。给定一组数据点，需要建立一个可以在类之间绘制线性边界的模型。逻辑回归就可以对训练数据派生的一组方程进行求解来提取边界。

详细步骤

(1) 下面看看用Python如何实现逻辑回归。我们使用logistic_regression.py文件作为参考。假设已经导入了需要使用的程序包，接下来创建一些带训练标记的样本数据：

```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt

X = np.array([[4, 7], [3.5, 8], [3.1, 6.2], [0.5, 1], [1, 2],
[1.2, 1.9], [6, 2], [5.7, 1.5], [5.4, 2.2]])
y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

这里假设一共有3个类。

(2) 初始化一个逻辑回归分类器：

```
classifier = linear_model.LogisticRegression(solver='liblinear', C=100)
```

前面的函数有一些输入参数需要设置，但是最重要的两个参数是solver和C。参数solver用于设置求解系统方程的算法类型，参数C表示正则化强度，数值越小，表示正则化强度越高。

(3) 接下来训练分类器：

```
classifier.fit(X, y)
```

(4) 画出数据点和边界：

```
plot_classifier(classifier, X, y)
```

需要定义如下画图函数：

```
def plot_classifier(classifier, X, y):
    # 定义图形的取值范围
    x_min, x_max = min(X[:, 0]) - 1.0, max(X[:, 0]) + 1.0
    y_min, y_max = min(X[:, 1]) - 1.0, max(X[:, 1]) + 1.0
```

预测值表示我们在图形中想要使用的数值范围，通常是从最小值到最大值。我们增加了一些余量（buffer），例如上面代码中的1.0。

(5) 为了画出边界，还需要利用一组网格（grid）数据求出方程的值，然后把边界画出来。下面继续定义网格：

```
    # 设置网格数据的步长
    step_size = 0.01

    # 定义网格
    x_values, y_values = np.meshgrid(np.arange(x_min, x_max, step_size),
np.arange(y_min, y_max, step_size))
```

变量x_values和y_values包含求解方程数值的网格点。

(6) 计算出分类器对所有数据点的分类结果：

```
    # 计算分类器输出结果
    mesh_output = classifier.predict(np.c_[x_values.ravel(), y_values.ravel()])

    # 数组维度变形
    mesh_output = mesh_output.reshape(x_values.shape)
```

(7) 用彩色区域画出各个类型的边界：

```
    # 用彩图画出分类结果
    plt.figure()

    # 选择配色方案
    plt.pcolormesh(x_values, y_values, mesh_output, cmap=plt.cm.gray)
```

这基本算是一个三维画图器，既可以画二维数据点，又可以用色彩清单（color scheme）表示不同区域的相关属性。你可以在http://matplotlib.org/examples/color/colormaps_reference.html 找到所有的色彩清单。

(8) 接下来再把训练数据点画在图上：

```
    plt.scatter(X[:, 0], X[:, 1], c=y, s=80, edgecolors='black', linewidth=1,
cmap=plt.cm.Paired)

    # 设置图形的取值范围
```

```
plt.xlim(x_values.min(), x_values.max())
plt.ylim(y_values.min(), y_values.max())

# 设置X轴与Y轴
plt.xticks((np.arange(int(min(X[:, 0])-1), int(max(X[:, 0])+1), 1.0)))
plt.yticks((np.arange(int(min(X[:, 1])-1), int(max(X[:, 1])+1), 1.0)))

plt.show()
```

其中，plt.scatter把数据点画在二维图上。X[:, 0]表示0轴（X轴）的坐标值，X[:, 1]表示1轴（Y轴）的坐标值。c=y表示颜色的使用顺序。用目标标记映射cmap的颜色表。我们肯定希望不同的标记使用不同的颜色，因此，用y作为映射。坐标轴的取值范围由plt.xlim和plt.ylim确定。为了标记坐标轴的数值，需要使用plt.xticks和plt.yticks。在坐标轴上标出坐标值，就可以直观地看出数据点的位置。在前面的代码中，我们希望坐标轴的最大值与最小值之前的刻度是单位刻度，还希望这些刻度值是整数，因此用int()函数对最值取整。

(9) 运行代码，就可以看到如图2-3所示的输出结果。

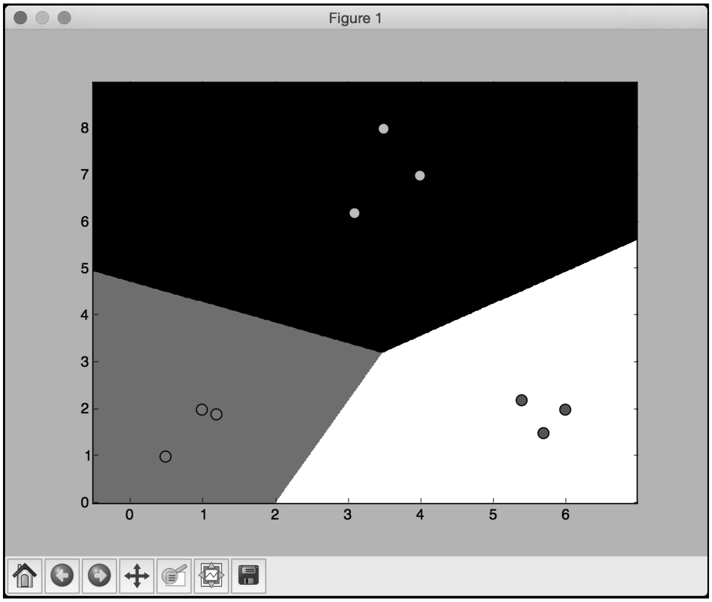


图 2-3

(10) 下面看看参数C对模型的影响。参数C表示对分类错误（misclassification）的惩罚值（penalty）。如果把参数C设置为1.0，会得到如图2-4所示的结果。

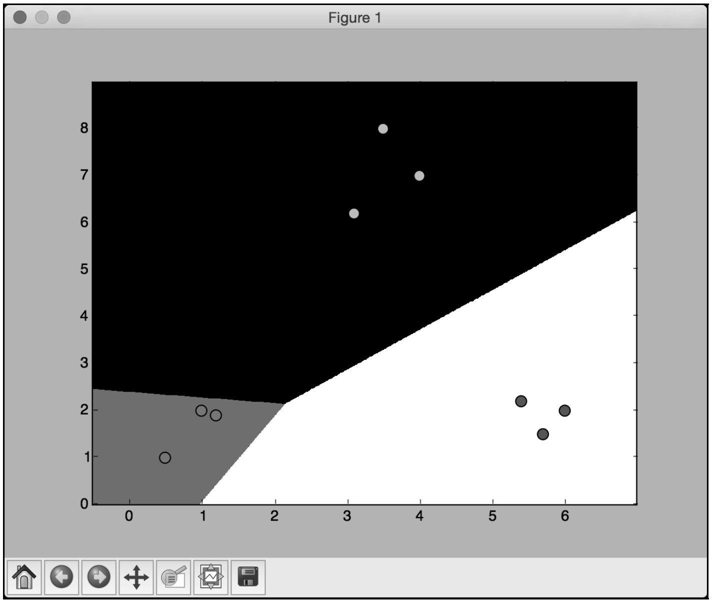


图　2-4

(11) 如果把参数C设置为10000，会得到如图2-5所示的结果。

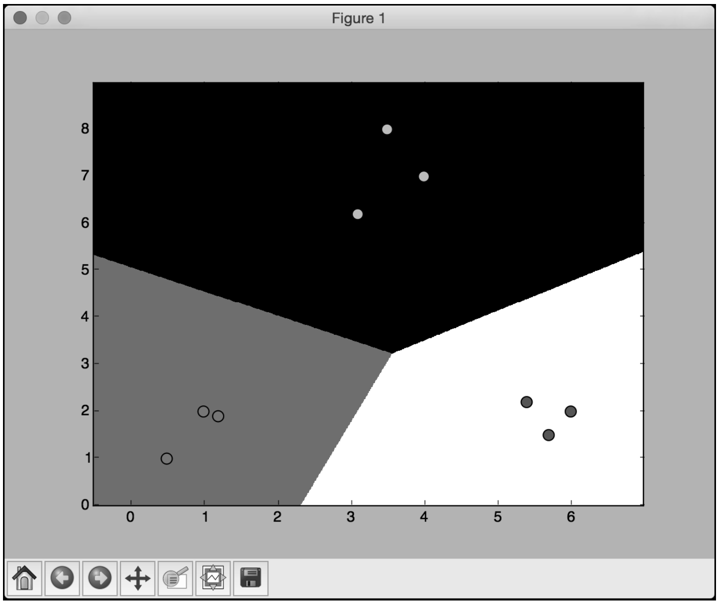


图　2-5

随着参数C的不断增大，分类错误的惩罚值越高。因此，各个类型的边界更优。

2.4　建立朴素贝叶斯分类器

朴素贝叶斯分类器是用贝叶斯定理进行建模的监督学习分类器。下面看看如何建立一个朴素贝叶斯分类器。

详细步骤

(1) 我们使用naive_bayes.py文件作为参考。首先导入两个程序包：

```
from sklearn.naive_bayes import GaussianNB
from logistic_regression import plot_classifier
```

(2) 下载的示例代码中有一个data_multivar.txt文件，里面包含了将要使用的数据，每一行数据都是由逗号分隔符分割的数值。从文件中加载数据：

```
input_file = 'data_multivar.txt'

X = []
y = []
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = [float(x) for x in line.split(',')]
        X.append(data[:-1])
        y.append(data[-1])

X = np.array(X)
y = np.array(y)
```

我们已经把输入数据和标记分别加载到变量X和y中了。

(3) 下面建立一个朴素贝叶斯分类器：

```
classifier_gaussiannb = GaussianNB()
classifier_gaussiannb.fit(X, y)
y_pred = classifier_gaussiannb.predict(X)
```

GaussianNB函数指定了正态分布朴素贝叶斯模型（Gaussian Naive Bayes model）。

(4) 接下来计算分类器的准确性：

```
accuracy = 100.0 * (y == y_pred).sum() / X.shape[0]
print "Accuracy of the classifier =", round(accuracy, 2), "%"
```

(5) 画出数据点和边界：

```
plot_classifier(classifier_gaussiannb, X, y)
```

可以看到如图2-6所示的图形。

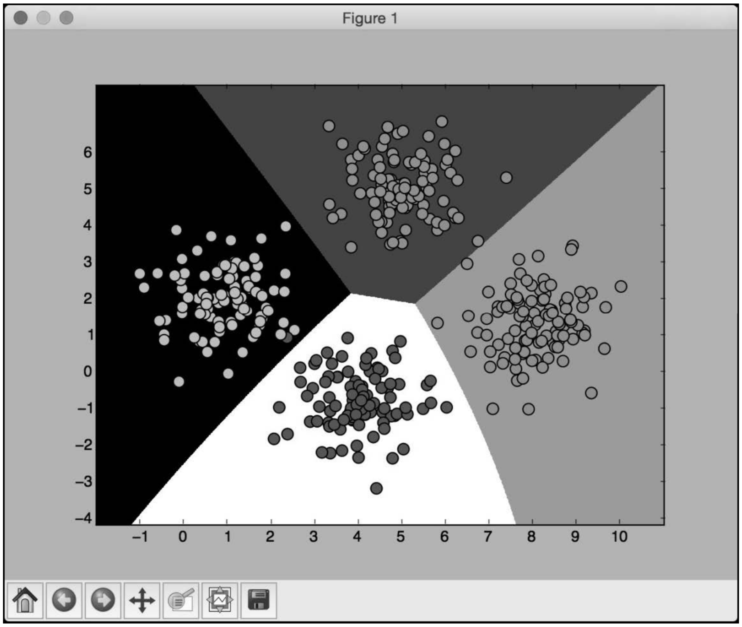


图　2-6

从图2-6中可以发现，这里的边界没有严格地区分所有数据点。在前面这个例子中，我们是对所有的数据进行训练。机器学习的一条最佳实践是用没有重叠（nonoverlapping）的数据进行训练和测试。理想情况下，需要一些尚未使用的数据进行测试，可以方便准确地评估模型在未知数据上的执行情况。scikit-learn有一个方法可以非常好地解决这个问题，我们将在下一节介绍它。

2.5　将数据集分割成训练集和测试集

本节一起来看看如何将数据合理地分割成训练数据集和测试数据集。

详细步骤

(1) 增加下面的代码片段到上一节的Python文件中：

```
from sklearn import cross_validation

X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y,
test_size=0.25, random_state=5)

classifier_gaussiannb_new = GaussianNB()
classifier_gaussiannb_new.fit(X_train, y_train)
```

这里，我们把参数`test_size`设置成`0.25`，表示分配了25%的数据给测试数据集。剩下75%

的数据将用于训练数据集。

(2) 用分类器对测试数据进行测试：

```
y_test_pred = classifier_gaussiannb_new.predict(X_test)
```

(3) 计算分类器的准确性：

```
accuracy = 100.0 * (y_test == y_test_pred).sum() / X_test.shape[0]
print "Accuracy of the classifier =", round(accuracy, 2), "%"
```

(4) 画出测试数据的数据点及其边界：

```
plot_classifier(classifier_gaussiannb_new, X_test, y_test)
```

(5) 可以看到如图2-7所示的图形。

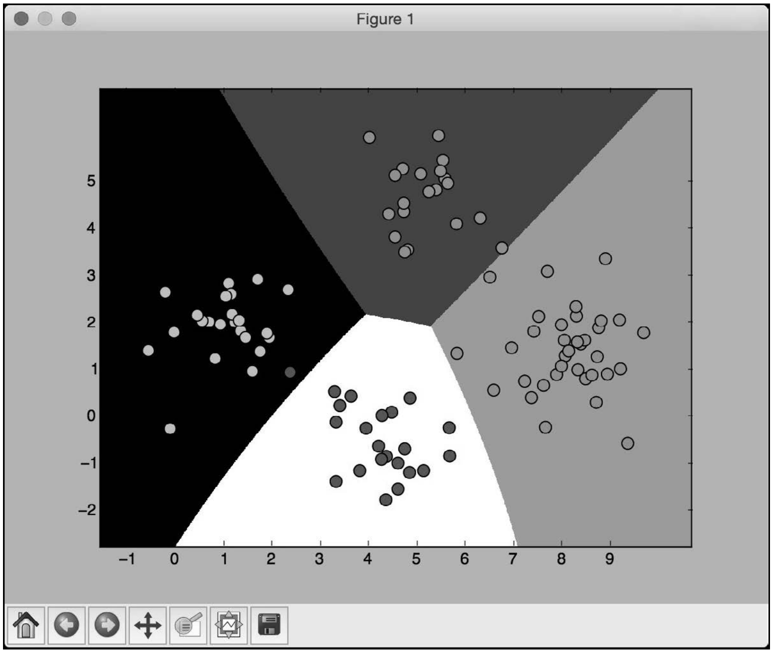


图　2-7

2.6　用交叉验证检验模型准确性

交叉验证是机器学习的重要概念。在上一节中，我们把数据分成了训练数据集和测试数据集。然而，为了能够让模型更加稳定，还需要用数据集的不同子集进行反复的验证。如果只是对特定的子集进行微调，最终可能会过度拟合（overfitting）模型。过度拟合是指模型在已知数据集上

拟合得超级好，但是一遇到未知数据就挂了。我们真正想要的，是让机器学习模型能够适用于未知数据。

2.6.1　准备工作

介绍如何实现交叉验证之前，先讨论一下性能指标。当处理机器学习模型时，通常关心3个指标：精度（precision）、召回率（recall）和F1得分（F1 score）。可以用参数评分标准（parameter scoring）获得各项指标的得分。精度是指被分类器正确分类的样本数量占分类器总分类样本数量的百分比（分类器分类结果中，有一些样本分错了）。召回率是指被应正确分类的样本数量占某分类总样本数量的百分比（有一些样本属于某分类，但分类器却没有分出来）。

假设数据集有100个样本，其中有82个样本是我们感兴趣的，现在想用分类器选出这82个样本。最终，分类器选出了73个样本，它认为都是我们感兴趣的。在这73个样本中，其实只有65个样本是我们感兴趣的，剩下的8个样本我们不感兴趣，是分类器分错了。可以如下方法计算分类器的精度：

- 分类正确的样本数量 = 65
- 总分类样本数量 = 73
- 精度 = 65 / 73 = 89.04%

召回率的计算过程如下：

- 数据集中我们感兴趣的样本数量 = 82
- 分类正确的样本数量 = 65
- 召回率 = 65 / 82 = 79.26%

一个给力的机器学习模型需要同时具备良好的精度和召回率。这两个指标是二律背反的，一个指标达到100%，那么另一个指标就会非常差！我们需要保持两个指标能够同时处于合理高度。为了量化两个指标的均衡性，引入了F1得分指标，是精度和召回率的合成指标，实际上是精度和召回率的调和均值（harmonic mean）：

$$\text{F1 得分} = 2 \times \text{精度} \times \text{召回率} / (\text{精度} + \text{召回率})$$

上面示例中F1得分的计算过程如下：

$$\text{F1 得分} = 2 \times 0.89 \times 0.79 / (0.89 + 0.79) = 0.8370$$

2.6.2　详细步骤

(1) 下面看看如何实现交叉验证，并提取性能指标。首先计算精度：

```
num_validations = 5
accuracy = cross_validation.cross_val_score(classifier_gaussiannb,
        X, y,scoring='accuracy', cv=num_validations)
print "Accuracy: " + str(round(100*accuracy.mean(), 2)) + "%"
```

(2) 用前面的方程分别计算精度、召回率和F1得分：

```
f1 = cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='f1_weighted', cv=num_validations)
print "F1: " + str(round(100*f1.mean(), 2)) + "%"

precision = cross_validation.cross_val_score(classifier_ gaussiannb,
        X, y, scoring='precision_weighted', cv=num_validations)
print "Precision: " + str(round(100*precision.mean(), 2)) + "%"

recall = cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='recall_weighted', cv=num_validations)
print "Recall: " + str(round(100*recall.mean(), 2)) + "%"
```

2.7　混淆矩阵可视化

混淆矩阵（confusion matrix）是理解分类模型性能的数据表，它有助于我们理解如何把测试数据分成不同的类。当想对算法进行调优时，就需要在对算法做出改变之前了解数据的错误分类情况。有些分类效果比其他分类效果更差，混淆矩阵可以帮助我们理解这些问题。先看看如图2-8所示的混淆矩阵。

	Predicted class 0	Predicted class 1	Predicted class 2
True class 0	45	4	3
True class 1	11	56	2
True class 2	5	6	49

图　2-8

在图2-8中，我们可以看出不同类型的分类数据。理想情况下，我们希望矩阵非对角线元素都是0，这是最完美的分类结果。先看看class 0，一共52个样本属于class 0。如果对第一行数据求和，总数就是52。但是现在，只有45个样本被正确地预测出来，分类器说另外4个样本属于class 1，还有3个样本属于class 2。用同样的思路分析另外两行数据，有意思的是，class 1里面有11个样本被错误地预测成了class 0，占到了class 1总数的16%。这就是模型需要优化的切入点。

详细步骤

(1) 我们用confusion_matrix.py文件作为参考。首先看看如何从数据中提取混淆矩阵：

```
from sklearn.metrics import confusion_matrix
y_true = [1, 0, 0, 2, 1, 0, 3, 3, 3]
y_pred = [1, 1, 0, 2, 1, 0, 1, 3, 3]
confusion_mat = confusion_matrix(y_true, y_pred)
plot_confusion_matrix(confusion_mat)
```

这里用了一些样本数据，一共有4种类型，取值范围是0~3，也列出了预测的标记类型。用confusion_matrix方法提取混淆矩阵，然后把它画出来。

(2) 继续定义混淆矩阵的画图函数：

```
# 显示混淆矩阵
def plot_confusion_matrix(confusion_mat):
    plt.imshow(confusion_mat, interpolation='nearest', cmap=plt.cm.Paired)
    plt.title('Confusion matrix')
    plt.colorbar()
    tick_marks = np.arange(4)
    plt.xticks(tick_marks, tick_marks)
    plt.yticks(tick_marks, tick_marks)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

这里用imshow函数画混淆矩阵，其他函数都非常简单，只使用相关函数设置了图形的标题、颜色栏、刻度和标签。参数tick_marks的取值范围是0~3，因为数据集中有4个标记类型。np.arange函数会生成一个numpy数组。

(3) 运行代码，可以看到如图2-9所示的图形。

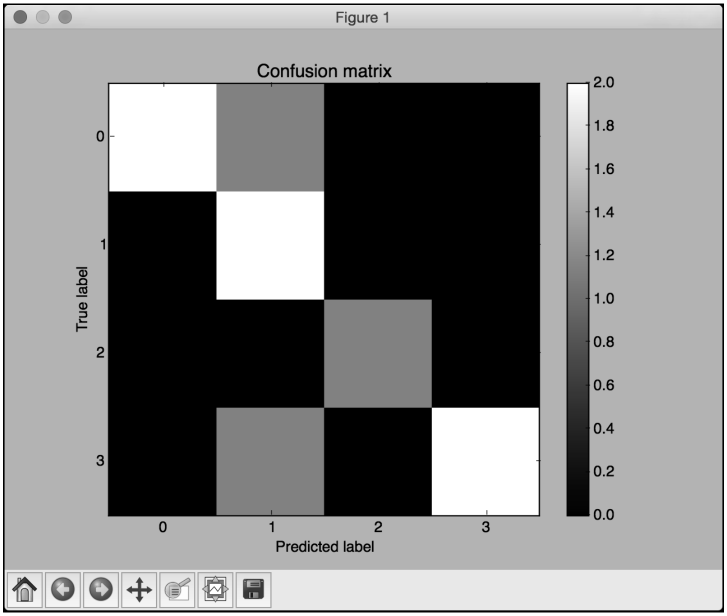


图　2-9

从图2-9中可以看出，对角线的颜色很亮，我们希望它们越亮越好。黑色区域表示0。在非对角线的区域有一些灰色区域，表示分类错误的样本量。例如，当样本真实标记类型是0，而预测标记类型是1时，就像在第一行的第二格看到的那样。事实上，所有的错误分类都属于 class-1，因为第二列有3个不为0的格子。这在图2-9中显示得一目了然。

2.8　提取性能报告

也可以直接用scikit-learn打印精度、召回率和F1得分。接下来看看如何实现。

详细步骤

(1) 在一个新的Python文件中加入下面的代码：

```
from sklearn.metrics import classification_report
y_true = [1, 0, 0, 2, 1, 0, 3, 3, 3]
y_pred = [1, 1, 0, 2, 1, 0, 1, 3, 3]
target_names = ['Class-0', 'Class-1', 'Class-2', 'Class-3']
print(classification_report(y_true, y_pred, target_names=target_names))
```

(2) 运行代码，可以在命令行工具中看到如图2-10所示的结果。

	precision	recall	f1-score	support
Class-0	1.00	0.67	0.80	3
Class-1	0.50	1.00	0.67	2
Class-2	1.00	1.00	1.00	1
Class-3	1.00	0.67	0.80	3
avg / total	0.89	0.78	0.79	9

图　2-10

不需要单独计算各个指标，可以直接用这个函数从模型中提取所有统计值。

2.9　根据汽车特征评估质量

接下来看看如何用分类技术解决现实问题。我们将用一个包含汽车多种细节的数据集，例如车门数量、后备箱大小、维修成本等，来确定汽车的质量。分类的目的是把车辆的质量分成4种类型：不达标、达标、良好、优秀。

2.9.1　准备工作

你可以从https://archive.ics.uci.edu/ml/datasets/Car+Evaluation下载数据集。

你需要把数据集中的每个值看成是字符串。考虑数据集中的6个属性，其取值范围是这样的：

- `buying`：取值范围是`vhigh`、`high`、`med`、`low`；
- `maint`：取值范围是`vhigh`、`high`、`med`、`low`；
- `doors`：取值范围是`2`、`3`、`4`、`5`等；
- `persons`：取值范围是`2`、`4`等；
- `lug_boot`：取值范围是`small`、`med`、`big`；
- `safety`：取值范围是`low`、`med`、`high`。

考虑到每一行都包含字符串属性，需要假设所有特征都是字符串，并设置分类器。在上一章中，我们用随机森林建立过回归器，这里再用随机森林建立分类器。

2.9.2　详细步骤

(1) 参考car.py文件中的源代码。首先导入两个软件包：

```
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
```

(2) 加载数据集：

```
input_file = 'path/to/dataset/car.data.txt'

# 读取数据
X = []
count = 0

with open(input_file, 'r') as f:
    for line in f.readlines():
        data = line[:-1].split(',')
        X.append(data)

X = np.array(X)
```

每一行都包含由逗号分隔的单词列表。因此，我们解析输入文件，对每一行进行分割，然后将该列表附加到主数据。我们忽略每一行最后一个字符，因为那是一个换行符。由于Python程序包只能处理数值数据，所以需要把这些属性转换成程序包可以理解的形式。

(3) 在上一章中，我们介绍过标记编码。下面可以用这个技术把字符串转换成数值：

```
# 将字符串转化为数值
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    label_encoder.append(preprocessing.LabelEncoder())
    X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

由于每个属性可以取有限数量的数值，所以可以用标记编码器将它们转换成数字。我们需要为不同的属性使用不同的标记编码器，例如，`lug_boot`属性可以取3个不同的值，需要建立一个懂得给这3个属性编码的标记编码器。每一行的最后一个值是类，将它赋值给变量`y`。

(4) 接下来训练分类器：

```
# 建立随机森林分类器
params = {'n_estimators': 200, 'max_depth': 8, 'random_state': 7}
classifier = RandomForestClassifier(**params)
classifier.fit(X, y)
```

你可以改变`n_estimators`和`max_depth`参数的值，观察它们如何改变分类器的准确性。我们将用一个标准化的方法处理参数选择问题。

(5) 下面进行交叉验证：

```
# 交叉验证
from sklearn import cross_validation
```

```
accuracy = cross_validation.cross_val_score(classifier,
        X, y, scoring='accuracy', cv=3)
print "Accuracy of the classifier: " + str(round(100*accuracy. mean(), 2)) + "%"
```

一旦训练好分类器，我们就需要知道它是如何执行的。我们用三折交叉验证（three-fold cross-validation，把数据分3组，轮换着用其中两组数据验证分类器）来计算分类器的准确性。

(6) 建立分类器的主要目的就是要用它对孤立的和未知的数据进行分类。下面用分类器对一个单一数据点进行分类：

```
# 对单一数据示例进行编码测试
input_data = ['vhigh', 'vhigh', '2', '2', 'small', 'low']
input_data_encoded = [-1] * len(input_data)
for i,item in enumerate(input_data):
    input_data_encoded[i] = int(label_encoder[i].transform(input_data[i]))

input_data_encoded = np.array(input_data_encoded)
```

第一步是把数据转换成数值类型。需要使用之前训练分类器时使用的标记编码器，因为我们需要保持数据编码规则的前后一致。如果输入数据点里出现了未知数据，标记编码器就会出现异常，因为它不知道如何对这些数据进行编码。例如，如果你把列表中的第一个值vhigh改成abcd，那么标记编码器就不知道如何编码了，因为它不知道怎么处理这个字符串。这就像是错误检查，看看输入数据点是否有效。

(7) 现在可以预测出数据点的输出类型了：

```
# 预测并打印特定数据点的输出
output_class = classifier.predict(input_data_encoded)
print "Output class:", label_encoder[-1].inverse_transform(output_class)[0]
```

我们用predict方法估计输出类型。如果输出被编码的输出标记，那么它对我们没有任何意义。因此，用inverse_transform方法对标记进行解码，将它转换成原来的形式，然后打印输出类。

2.10　生成验证曲线

前面用随机森林建立了分类器，但是并不知道如何定义参数。本节来处理两个参数：n_estimators和max_depth参数。它们被称为**超参数**（hyperparameters），分类器的性能是由它们决定的。当改变超参数时，如果可以看到分类器性能的变化情况，那就再好不过了。这就是验证曲线的作用。这些曲线可以帮助理解每个超参数对训练得分的影响。基本上，我们只对感兴趣的超参数进行调整，其他参数可以保持不变。下面将通过可视化图片演示超参数的变化对训练得分的影响。

详细步骤

(1) 打开上一节的Python文件，加入以下代码：

```
# 验证曲线

from sklearn.learning_curve import validation_curve

classifier = RandomForestClassifier(max_depth=4, random_state=7)
parameter_grid = np.linspace(25, 200, 8).astype(int)
train_scores, validation_scores = validation_curve(classifier, X, y, "n_estimators",
parameter_grid, cv=5)
print "\n##### VALIDATION CURVES #####"
print "\nParam: n_estimators\nTraining scores:\n", train_scores
print "\nParam: n_estimators\nValidation scores:\n", validation_ scores
```

在这个示例中，我们通过固定`max_depth`参数的值来定义分类器。我们想观察评估器数量对训练得分的影响，于是用`parameter_grid`定义了搜索空间。评估器数量会在25~200之间每隔8个数迭代一次，获得模型的训练得分和验证得分。

(2) 运行代码，可以在命令行工具中看到如图2-11所示的结果。

```
##### VALIDATION CURVES #####

Param: n_estimators
Training scores:
[[ 0.80680174  0.80824891  0.80752533  0.80463097  0.81358382]
 [ 0.79522431  0.80535456  0.81041968  0.8089725   0.81069364]
 [ 0.80101302  0.80680174  0.81114327  0.81476122  0.8150289 ]
 [ 0.8024602   0.80535456  0.81186686  0.80752533  0.80346821]
 [ 0.80028944  0.80463097  0.81114327  0.80824891  0.81069364]
 [ 0.80390738  0.80535456  0.81041968  0.80969609  0.81647399]
 [ 0.80390738  0.80463097  0.81114327  0.81476122  0.81719653]
 [ 0.80390738  0.80607815  0.81114327  0.81403763  0.81647399]]

Param: n_estimators
Validation scores:
[[ 0.71098266  0.76589595  0.72543353  0.76300578  0.75290698]
 [ 0.71098266  0.75433526  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.72254335  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.71387283  0.71965318  0.75722543  0.72674419]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74566474  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.75144509  0.71965318  0.75722543  0.74127907]]
```

图 2-11

(3) 把数据画成图形：

```
# 画出曲线图
plt.figure()
plt.plot(parameter_grid, 100*np.average(train_scores, axis=1), color='black')
plt.title('Training curve')
```

```
plt.xlabel('Number of estimators')
plt.ylabel('Accuracy')
plt.show()
```

(4) 得到的图形如图2-12所示。

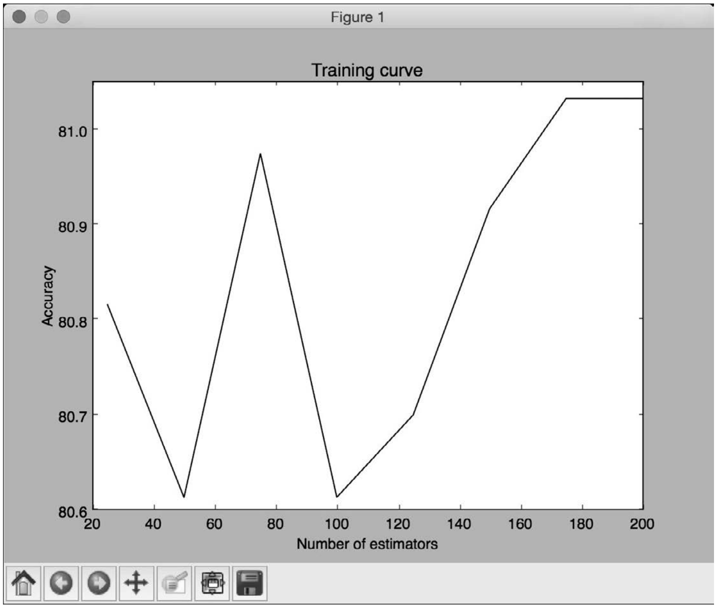


图　2-12

(5) 用类似的方法对max_depth参数进行验证：

```
classifier = RandomForestClassifier(n_estimators=20, random_ state=7)
parameter_grid = np.linspace(2, 10, 5).astype(int)
train_scores, valid_scores = validation_curve(classifier, X, y, "max_depth",
        parameter_grid, cv=5)
print "\nParam: max_depth\nTraining scores:\n", train_scores
print "\nParam: max_depth\nValidation scores:\n", validation_ scores
```

我们把n_estimators参数固定为20，看看max_depth参数变化对性能的影响。命令行工具的输出结果如图2-13所示。

(6) 把数据画成图形：

```
# 画出曲线图
plt.figure()
plt.plot(parameter_grid, 100*np.average(train_scores, axis=1), color='black')
plt.title('Validation curve')
plt.xlabel('Maximum depth of the tree')
plt.ylabel('Accuracy')
plt.show()
```

```
Param: max_depth
Training scores:
[[ 0.71852388  0.70043415  0.70043415  0.70043415  0.69942197]
 [ 0.80607815  0.80535456  0.80752533  0.79450072  0.81069364]
 [ 0.90665702  0.91027496  0.92836469  0.89797395  0.90679191]
 [ 0.97467438  0.96743849  0.97105644  0.97829233  0.96820809]
 [ 0.99421129  0.99782923  0.99782923  0.99855282  0.99421965]]

Param: max_depth
Validation scores:
[[ 0.71098266  0.76589595  0.72543353  0.76300578  0.75290698]
 [ 0.71098266  0.75433526  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.72254335  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.71387283  0.71965318  0.75722543  0.72674419]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74566474  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.75144509  0.71965318  0.75722543  0.74127907]]
```

图 2-13

(7) 运行代码，可以看到如图2-14所示的图形。

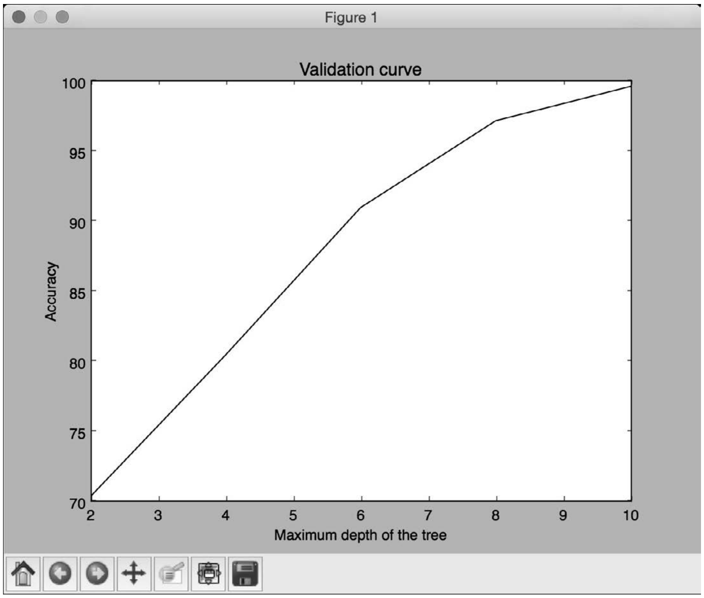


图 2-14

2.11 生成学习曲线

学习曲线可以帮助我们理解训练数据集的大小对机器学习模型的影响。当遇到计算能力限制时，这一点非常有用。下面改变训练数据集的大小，把学习曲线画出来。

详细步骤

(1) 打开上一节的Python文件，加入以下代码：

```
# 学习曲线

from sklearn.learning_curve import learning_curve

classifier = RandomForestClassifier(random_state=7)

parameter_grid = np.array([200, 500, 800, 1100])
train_sizes, train_scores, validation_scores = learning_ curve(classifier,
        X, y, train_sizes=parameter_grid, cv=5)
print "\n##### LEARNING CURVES #####"
print "\nTraining scores:\n", train_scores
print "\nValidation scores:\n", validation_scores
```

我们想分别用200、500、800、1100的训练数据集的大小测试模型的性能指标。我们把`learning_curve`方法中的`cv`参数设置为`5`，就是用五折交叉验证。

(2) 运行代码，可以在命令行工具中看到如图2-15所示的结果。

```
##### LEARNING CURVES #####

Training scores:
[[ 1.          1.          1.          1.          1.        ]
 [ 1.          1.          0.998       0.998       0.998     ]
 [ 0.99875     0.99875     0.99875     0.99875     0.99875   ]
 [ 0.99909091  0.99545455  0.99909091  0.99818182  0.99818182]]

Validation scores:
[[ 0.69942197  0.69942197  0.69942197  0.69942197  0.70348837]
 [ 0.75433526  0.65028902  0.76878613  0.76589595  0.70348837]
 [ 0.70520231  0.78612717  0.52312139  0.76878613  0.77034884]
 [ 0.6416185   0.75722543  0.64450867  0.75433526  0.76744186]]
```

图　2-15

(3) 把数据画成图形：

```
# 画出曲线图
plt.figure()
plt.plot(parameter_grid, 100*np.average(train_scores, axis=1), color='black')
plt.title('Learning curve')
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.show()
```

(4) 得到的图形如图2-16所示。

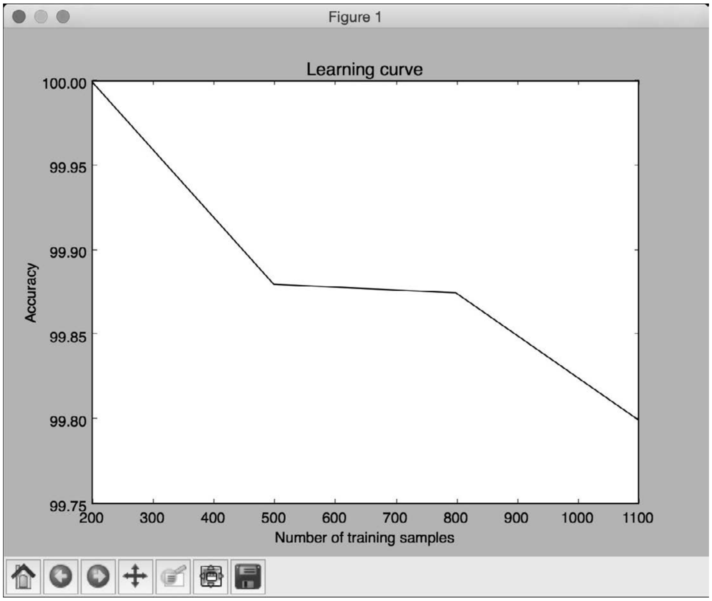


图 2-16

虽然训练数据集的规模越小，仿佛训练准确性越高，但是它们很容易导致过度拟合。如果选择较大规模的训练数据集，就会消耗更多的资源。因此，训练数据集的规模选择也是一个需要结合计算能力进行综合考虑的问题。

2.12 估算收入阶层

本节将根据14个属性建立分类器评估一个人的收入等级。可能的输出类型是“高于50K”和“低于或等于50K”。这个数据集稍微有点复杂，里面的每个数据点都是数字和字符串的混合体。数值数据是有价值的，在这种情况下，不能用标记编码器进行编码。需要设计一套既可以处理数值数据，也可以处理非数值数据的系统。我们将用美国人口普查收入数据集中的数据：https://archive.ics.uci.edu/ml/datasets/Census+Income 。

详细步骤

(1) 我们将用income.py文件作为参考，用朴素贝叶斯分类器解决问题。首先导入两个软件包：

```
from sklearn import preprocessing
from sklearn.naive_bayes import GaussianNB
```

(2) 加载数据集：

```
input_file = 'path/to/adult.data.txt'

# 读取数据
X = []
y = []
count_lessthan50k = 0
count_morethan50k = 0
num_images_threshold = 10000
```

(3) 我们将使用数据集中的20 000个数据点——每种类型10 000个，保证初始类型没有偏差。在模型训练时，如果你的大部分数据点都属于一个类型，那么分类器就会倾向于这个类型。因此，最好使用每个类型数据点数量相等的数据进行训练：

```
with open(input_file, 'r') as f:
    for line in f.readlines():
        if '?' in line:
            continue

        data = line[:-1].split(', ')

        if data[-1] == '<=50K' and count_lessthan50k < num_images_threshold:
            X.append(data)
            count_lessthan50k = count_lessthan50k + 1

        elif data[-1] == '>50K' and count_morethan50k < num_images_threshold:
            X.append(data)
            count_morethan50k = count_morethan50k + 1

        if count_lessthan50k >= num_images_threshold and count_morethan50k >=
num_images_threshold:
            break

X = np.array(X)
```

同样地，这也是一个带逗号分隔符的文件。我们还是像之前那样处理，把数据加载到变量X。

(4) 我们需要把字符串属性转换为数值数据，同时需要保留原有的数值数据：

```
# 将字符串转换为数值数据
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

isdigit()函数帮助我们判断一个属性是不是数值数据。我们把字符串数据转换为数值数

据，然后把所有的标记编码器保存在一个列表中，便于在后面处理未知数据时使用。

(5) 训练分类器：

```
# 建立分类器
classifier_gaussiannb = GaussianNB()
classifier_gaussiannb.fit(X, y)
```

(6) 把数据分割成训练数据集和测试数据集，方便后面获取性能指标：

```
# 交叉验证
from sklearn import cross_validation

X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y,
test_size=0.25, random_state=5)
classifier_gaussiannb = GaussianNB()
classifier_gaussiannb.fit(X_train, y_train)
y_test_pred = classifier_gaussiannb.predict(X_test)
```

(7) 提取性能指标：

```
# 计算分类器的F1得分
f1 = cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='f1_weighted', cv=5)
print "F1 score: " + str(round(100*f1.mean(), 2)) + "%"
```

(8) 接下来看看如何为单一数据点分类。我们需要把数据点转换成分类器可以理解的形式：

```
# 对单一数据示例进行编码测试
input_data = ['39', 'State-gov', '77516', 'Bachelors', '13', 'Never-married',
'Adm-clerical', 'Not-in-family', 'White', 'Male', '2174', '0', '40', 'United-States']
count = 0
input_data_encoded = [-1] * len(input_data)
for i,item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] = int(label_encoder[count].transform(input_data[i]))
        count = count + 1

input_data_encoded = np.array(input_data_encoded)
```

(9) 这样就可以进行分类了：

```
# 预测并打印特定数据点的输出结果
output_class = classifier_gaussiannb.predict(input_data_encoded)
print label_encoder[-1].inverse_transform(output_class)[0]
```

和之前的分类案例一样，我们用`predict`方法获取输出类型，然后用`inverse_transform`对标记进行解码，将它转换成原来的形式，然后在命令行工具中打印出来。

第 3 章

预测建模

在这一章，我们将介绍以下主题：

- 用SVM（Support Vector Machines，支持向量机）建立线性分类器
- 用SVM建立非线性分类器
- 解决类型数量不平衡问题
- 提取置信度
- 寻找最优超参数
- 建立事件预测器
- 估算交通流量

3.1 简介

预测建模（Predictive modeling）可能是数据分析中最吸引人的领域之一。近几年，由于大数据在各个垂直领域的蓬勃发展，预测建模备受关注。在数据挖掘领域，预测建模常用来预测未来趋势。

预测建模是一种用来预测系统未来行为的分析技术，它由一群能够识别独立输入变量与反馈目标关联关系的算法构成。我们根据观测值创建一个数学模型，然后用这个模型去预测未来发生的事情。

在预测建模中，需要收集已知的响应数据来训练模型。一旦模型建成，就可以用一些指标来检验它，然后用它预测未来值。可以通过许多种不同的算法来创建预测模型。本章将利用SVM来建立线性模型与非线性模型。

预测模型是用若干可能对系统行为产生影响的特征构建的。例如，如果要预测天气情况，就需要用气温、大气压、降雨量和其他的气象数据。类似地，当处理其他系统问题时，也需要先判断哪些因素可能会影响系统的行为，然后在训练模型之前把这些因素加入特征中。

3.2　用 SVM 建立线性分类器

SVM是用来构建分类器和回归器的监督学习模型。SVM通过对数学方程组求解，可以找出两组数据之间的最佳分割边界。如果你对SVM不太了解，可以从下面几个不错的教程开始学习：

- http://web.mit.edu/zoya/www/SVM.pdf
- http://www.support-vector.net/icml-tutorial.pdf
- http://www.svms.org/tutorials/Berwick2003.pdf

下面看看如何用SVM建立线性分类器。

3.2.1　准备工作

为了便于理解问题，先对数据进行可视化。我们将参考svm.py文件里已有的源代码。在建立SVM之前，先对数据进行直观的认识。我们将使用源代码文件夹里的data_multivar.txt文件。下面看看如何对数据进行可视化。首先创建一个Python文件，然后在文件中增加下面的代码：

```
import numpy as np
import matplotlib.pyplot as plt

import utilities

# 加载输入数据
input_file = 'data_multivar.txt'
X, y = utilities.load_data(input_file)
```

刚刚导入了需要的程序包，然后确定了输入文件的名称。接下来看看`load_data()`方法：

```
# 加载输入文件中的多变量数据
def load_data(input_file):
    X = []
    y = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data = [float(x) for x in line.split(',')]
            X.append(data[:-1])
            y.append(data[-1])

    X = np.array(X)
    y = np.array(y)

    return X, y
```

需要将数据分成类，如下所示：

```
class_0 = np.array([X[i] for i in range(len(X)) if y[i]==0])
class_1 = np.array([X[i] for i in range(len(X)) if y[i]==1])
```

数据分成类之后，我们把它们画出来：

```
plt.figure()
plt.scatter(class_0[:,0], class_0[:,1], facecolors='black', edgecolors='black',
marker='s')
plt.scatter(class_1[:,0], class_1[:,1], facecolors='None', edgecolors='black',
marker='s')
plt.title('Input data')
plt.show()
```

运行代码，可以看到如图3-1所示的图形。

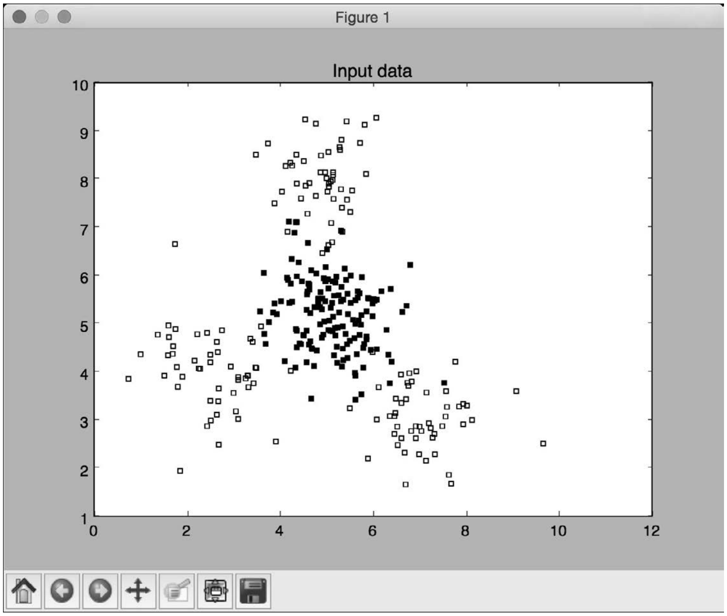


图　3-1

图3-1由两种类型的数据点构成——**实心方块**和**空心方块**。用机器学习的术语说就是，我们的数据由两个类型组成。我们的目标就是要建立一个可以将实心方块和空心方块分开的模型。

3.2.2　详细步骤

(1) 我们需要将数据集分割成训练数据集和测试数据集。在同样的Python文件中加入以下代码：

```
# 分割数据集并用SVM训练模型
from sklearn import cross_validation
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y,
test_size=0.25, random_state=5)
```

(2) 用线性核函数（linear kernel）初始化一个SVM对象。如果你不了解核函数的概念，请参考http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html。在文件中加入以下代码：

```
params = {'kernel': 'linear'}
classifier = SVC(**params)
```

(3) 现在可以训练线性SVM分类器了：

```
classifier.fit(X_train, y_train)
```

(4) 现在可以看到分类器是如何执行的：

```
utilities.plot_classifier(classifier, X_train, y_train, 'Training dataset')
plt.show()
```

(5) 运行代码，可以看到如图3-2所示的图形。

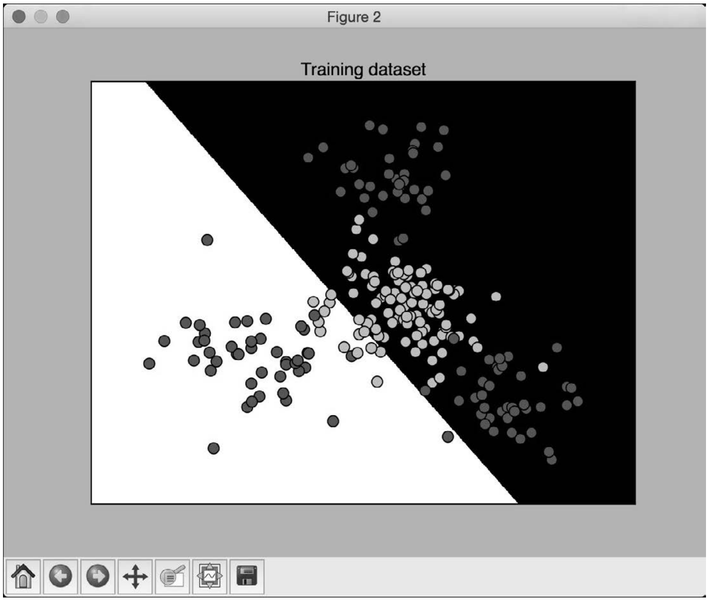


图 3-2

plot_classifier函数和之前介绍的画图函数一样，只是额外增加了两点内容。你可以在utilities.py文件里查看所有细节。

(6) 接下来看看分类器对测试数据集的执行。在Python文件中增加下面的代码：

```
y_test_pred = classifier.predict(X_test)
utilities.plot_classifier(classifier, X_test, y_test, 'Test dataset')
plt.show()
```

(7) 运行代码，可以看到如图3-3所示的图形。

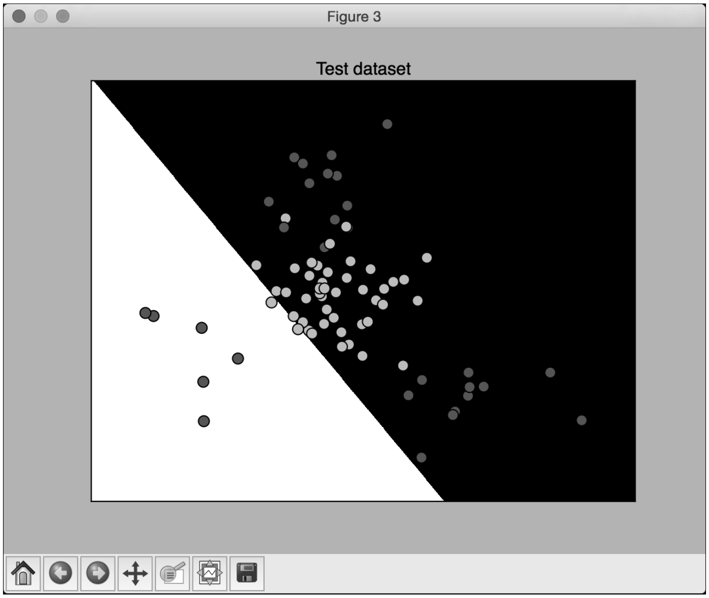


图　3-3

(8) 接下来计算训练数据集的准确性。在Python文件中增加下面的代码：

```
from sklearn.metrics import classification_report

target_names = ['Class-' + str(int(i)) for i in set(y)]
print "\n" + "#"*30
print "\nClassifier performance on training dataset\n"
print classification_report(y_train, classifier.predict(X_train),
target_names=target_names)
print "#"*30 + "\n"
```

(9) 运行代码，可以在命令行工具中看到如图3-4所示的结果。

```
##############################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.55      0.88      0.68       105
    Class-1       0.78      0.38      0.51       120

avg / total       0.67      0.61      0.59       225

##############################
```

图　3-4

(10) 最后看看分类器为测试数据集生成的分类报告：

```
print "#"*30
print "\nClassification report on test dataset\n"
print classification_report(y_test, y_test_pred, target_names=target_names)
print "#"*30 + "\n"
```

(11) 运行代码，可以在命令行工具中看到如图3-5所示的结果。

```
##############################

Classification report on test dataset

             precision    recall  f1-score   support

    Class-0       0.64      0.96      0.77        45
    Class-1       0.75      0.20      0.32        30

avg / total       0.69      0.65      0.59        75

##############################
```

图 3-5

从前面数据的可视化图中可以看出，实心方块完全是被空心方块包围着的，也就是说两种类型的数据不是线性可分的。我们无法画出一条可以分离两种类型数据点的完美直线，因此需要尝试使用非线性分类器来分离这两种数据。

3.3 用 SVM 建立非线性分类器

SVM为建立非线性分类器提供了许多选项，需要用不同的核函数建立非线性分类器。为了简单起见，考虑两种情况，当想要表示两种类型数据的曲线边界时，既可以用多项式函数，也可以用径向基函数（Radial Basis Function，RBF）。

详细步骤

(1) 对于第一种情况，可以用一个多项式核函数建立非线性分类器。在同样的Python文件中搜索下面的代码：

```
params = {'kernel': 'linear'}
```

将其替换成下面的代码：

```
params = {'kernel': 'poly', 'degree': 3}
```

这就表示我们用了一个三次多项式方程。如果增加方程的次数，就表示可以让曲线更加弯曲。但是，曲线越弯曲，意味着训练要花费的时间越长，因为计算强度更大。

(2) 运行代码，可以看到如图3-6所示的图形。

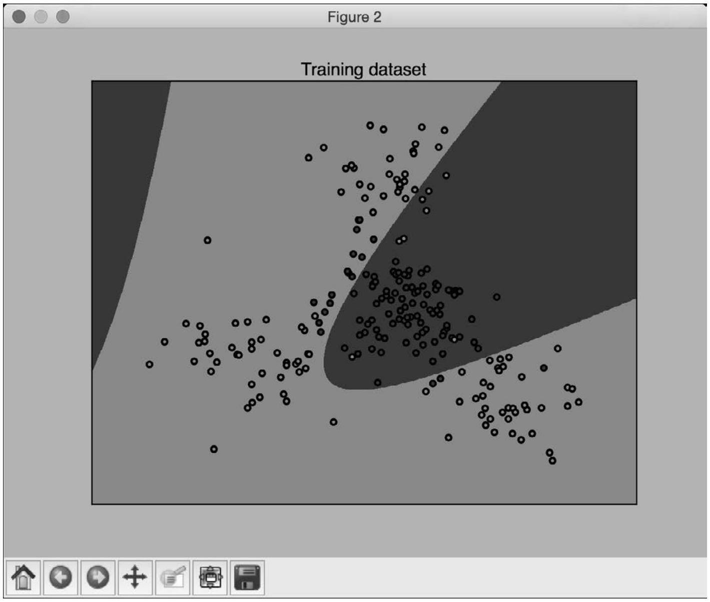


图　3-6

(3) 还可以在命令行工具中看到如图3-7所示的分类报告。



```
##############################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.92      0.84      0.88       105
    Class-1       0.87      0.93      0.90       120

avg / total       0.89      0.89      0.89       225

##############################
```

图　3-7

(4) 我们还可以用径向基函数建立非线性分类器。在同样的Python文件中搜索下面的代码：

```
params = {'kernel': 'poly', 'degree': 3}
```

将其替换成：

```
params = {'kernel': 'rbf'}
```

(5) 运行代码，可以看到如图3-8所示的图形。

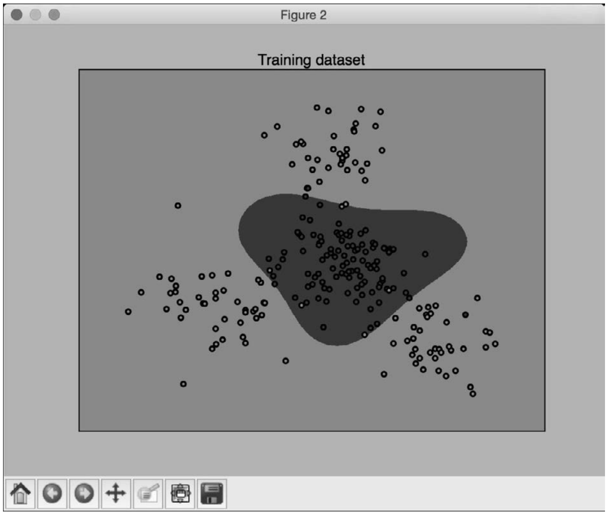


图 3-8

(6) 还可以在命令行工具中看到如图3-9所示的分类报告。


```
##############################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.95      0.98      0.97       105
    Class-1       0.98      0.96      0.97       120

avg / total       0.97      0.97      0.97       225

##############################
```

图 3-9

3.4 解决类型数量不平衡问题

到目前为止，我们处理的问题都是所有类型的数据点数量比较接近的情况，但是在真实世界中，我们不可能总能获取到这么均衡的数据集。有时，某一个类型的数据点数量可能比其他类型多很多，在这种条件下训练的分类器就会有偏差。边界线不会反映数据的真实特性，因为两种类型的数据点数量差别太大。因此，需要慎重考虑这种差异性，并想办法调和，才能保证分类器是不偏不倚的。

详细步骤

(1) 先加载数据：

```
input_file = 'data_multivar_imbalance.txt'
X, y = utilities.load_data(input_file)
```

(2) 接下来可视化这些数据。数据可视化的代码和前面章节中的可视化代码完全相同，可以参考svm_imbalance.py文件中的源代码。运行代码，可以看到如图3-10所示的图形。

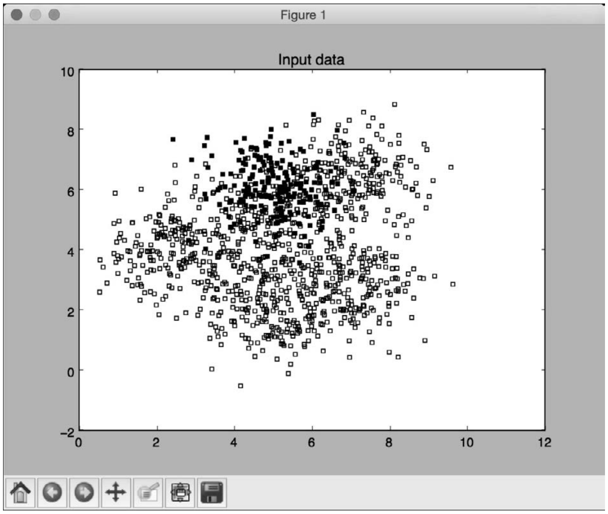


图　3-10

(3) 下面用线性核函数建立一个SVM分类器，代码和前面章节中的代码完全相同。运行代码，可以看到如图3-11所示的图形。

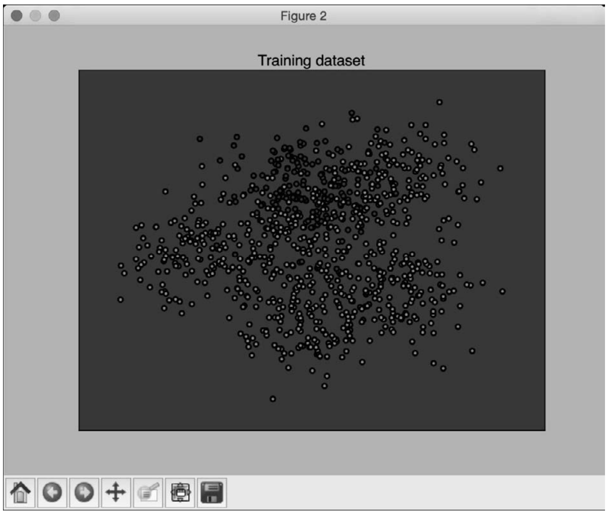


图　3-11

(4) 你可能会奇怪为什么没有边界线了。其实是因为分类器不能区分两种类型，导致`Class-0`的准确性是0%。可以在命令行工具中看到如图3-12所示的分类报告。

```
##############################

Classification report on test dataset

             precision    recall  f1-score   support

    Class-0       0.00      0.00      0.00        42
    Class-1       0.86      1.00      0.92       258

avg / total       0.74      0.86      0.80       300

##############################
```

图　3-12

从图3-12中也可以看出，`Class-0`的准确性是0%。

(5) 我们继续来解决这个问题。在Python文件中搜索下面的代码：

```
params = {'kernel': 'linear'}
```

将其替换成：

```
params = {'kernel': 'linear', 'class_weight': 'auto'}
```

参数`class_weight`的作用是统计不同类型数据点的数量，调整权重，让类型不平衡问题不影响分类效果。

(6) 运行代码，可以看到如图3-13所示的图形。

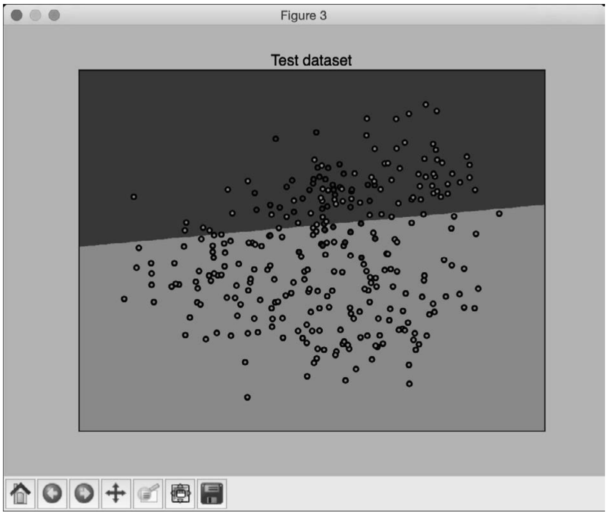


图　3-13

(7) 可以在命令行工具中看到如图3-14所示的分类报告。

```
##############################

Classification report on test dataset

             precision    recall  f1-score   support

    Class-0       0.29      0.76      0.42        42
    Class-1       0.95      0.70      0.80       258

avg / total       0.86      0.71      0.75       300

##############################
```

图　3-14

从图3-14中可以看到，`Class-0`的准确性不是0%了。

3.5　提取置信度

如果能够获取对未知数据进行分类的置信水平，那将会非常有用。当一个新的数据点被分类为某一个已知类别时，我们可以训练SVM来计算出输出类型的置信度。

详细步骤

(1) 读者可以参考svm_confidence.py文件中的源代码，这里只介绍核心部分。首先，定义以下输入数据：

```
# 测量数据点与边界的距离
input_datapoints = np.array([[2, 1.5], [8, 9], [4.8, 5.2], [4, 4], [2.5, 7], [7.6, 2],
[5.4, 5.9]])
```

(2) 测量数据点到边界的距离：

```
print "\nDistance from the boundary:"
for i in input_datapoints:
    print i, '-->', classifier.decision_function(i)[0]
```

(3) 可以在命令行工具中看到如图3-15所示的结果。

```
Distance from the boundary:
[ 2.   1.5] --> 0.92489688282
[ 8.   9.] --> 0.642239002462
[ 4.8  5.2] --> -2.03541766793
[ 4.   4.] --> -0.07623172175
[ 2.5  7. ] --> 0.734559329252
[ 7.6  2. ] --> 1.09824378145
[ 5.4  5.9] --> -1.21145495531
```

图　3-15

(4) 到边界的距离向我们提供了一些关于数据点的信息，但是它并不能准确地告诉我们分类器能够输出某个类型的置信度有多大。为了解决这个问题，需要用到**概率输出**（Platt scaling）。概率输出是一种将不同类别的距离度量转换成概率度量的方法。读者可以通过http://fastml.com/classifier-calibration-with-platts-scaling-and-isotonic-regression的教程学习概率输出的内容。下面继续用概率输出来训练SVM：

```
# 测量置信度
params = {'kernel': 'rbf', 'probability': True}
classifier = SVC(**params)
```


参数`probability`用于告诉SVM训练的时候要计算出概率。

(5) 训练分类器：

```
classifier.fit(X_train, y_train)
```

(6) 计算输入数据点的置信度：

```
print "\nConfidence measure:"
for i in input_datapoints:
    print i, '-->', classifier.predict_proba(i)[0]
```

`predict_proba`函数用于测量置信值。

(7) 可以在命令行工具中看到如图3-16所示的结果。

```
Confidence measure:
[ 2.   1.5] --> [ 0.05126939  0.94873061]
[ 8.   9.] --> [ 0.11146888  0.88853112]
[ 4.8  5.2] --> [ 0.99728099  0.00271901]
[ 4.   4.] --> [ 0.51684952  0.48315048]
[ 2.5  7. ] --> [ 0.08697132  0.91302868]
[ 7.6  2. ] --> [ 0.03124531  0.96875469]
[ 5.4  5.9] --> [ 0.96844526  0.03155474]
```

图　3-16

(8) 再看看数据点与边界的位置：

```
utilities.plot_classifier(classifier, input_datapoints, [0]*len(input_datapoints),
'Input datapoints', 'True')
```

(9) 运行代码，可以看到如图3-17所示的图形。

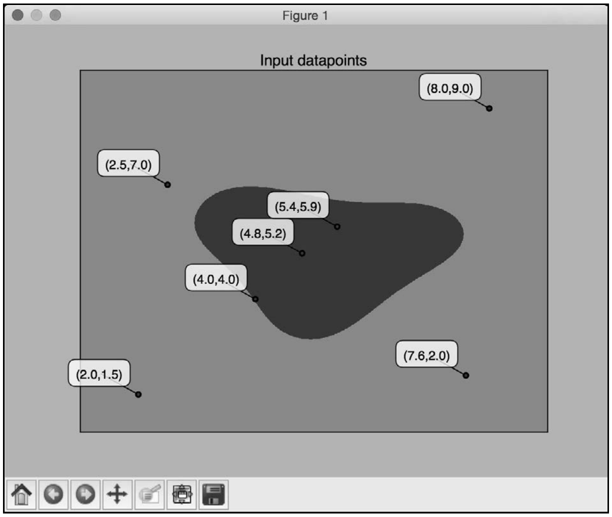


图　3-17

3.6　寻找最优超参数

就像上一章提到的，超参数对分类器的性能至关重要。本节我们看看如何为SVM获取最优的超参数。

详细步骤

(1) 读者可以参考perform_grid_search.py文件中的源代码，这里只介绍核心部分。这里将使用前面介绍过的交叉验证。加载完数据，并将数据分成训练数据集和测试数据集之后，向文件中加入下面的代码：

```
# 通过交叉检验设置参数
parameter_grid = [  {'kernel': ['linear'], 'C': [1, 10, 50, 600]},
                    {'kernel': ['poly'], 'degree': [2, 3]},
                    {'kernel': ['rbf'], 'gamma': [0.01, 0.001], 'C': [1, 10, 50, 600]},
                   ]
```

(2) 定义需要使用的指标：

```
metrics = ['precision', 'recall_weighted']
```

(3) 下面开始为每个指标搜索最优超参数：

```
for metric in metrics:
    print "\n#### Searching optimal hyperparameters for", metric
```

```
    classifier = grid_search.GridSearchCV(svm.SVC(C=1),
            parameter_grid, cv=5, scoring=metric)
    classifier.fit(X_train, y_train)
```

(4) 看看指标的得分：

```
print "\nScores across the parameter grid:"
for params, avg_score, _ in classifier.grid_scores_:
    print params, '-->', round(avg_score, 3)
```


(5) 打印出最好的参数集：

```
print "\nHighest scoring parameter set:", classifier.best_params_
```

(6) 运行代码，可以在命令行工具中看到如图3-18所示的结果。

```
#### Searching optimal hyperparameters for precision

Scores across the parameter grid:
{'kernel': 'linear', 'C': 1} --> 0.676
{'kernel': 'linear', 'C': 10} --> 0.676
{'kernel': 'linear', 'C': 50} --> 0.676
{'kernel': 'linear', 'C': 600} --> 0.676
{'kernel': 'poly', 'degree': 2} --> 0.872
{'kernel': 'poly', 'degree': 3} --> 0.872
{'kernel': 'rbf', 'C': 1, 'gamma': 0.01} --> 0.98
{'kernel': 'rbf', 'C': 1, 'gamma': 0.001} --> 0.533
{'kernel': 'rbf', 'C': 10, 'gamma': 0.01} --> 0.983
{'kernel': 'rbf', 'C': 10, 'gamma': 0.001} --> 0.543
{'kernel': 'rbf', 'C': 50, 'gamma': 0.01} --> 0.959
{'kernel': 'rbf', 'C': 50, 'gamma': 0.001} --> 0.806
{'kernel': 'rbf', 'C': 600, 'gamma': 0.01} --> 0.967
{'kernel': 'rbf', 'C': 600, 'gamma': 0.001} --> 0.983

Highest scoring parameter set: {'kernel': 'rbf', 'C': 10, 'gamma': 0.01}
```

图 3-18

(7) 从图3-18可以发现，模型搜索到了所有的最优超参数。在这个示例中，超参数是内核、C值和gamma的类型。模型将会尝试各种不同参数的组合来搜索最佳参数。接下来在测试数据集上做测试：

```
y_true, y_pred = y_test, classifier.predict(X_test)
print "\nFull performance report:\n"
print classification_report(y_true, y_pred)
```

(8) 运行代码，可以在命令行工具中看到如图3-19所示的结果。

```
Full performance report:

             precision    recall  f1-score   support

        0.0       0.92      0.98      0.95        45
        1.0       0.96      0.87      0.91        30

avg / total       0.94      0.93      0.93        75
```

图 3-19

3.7 建立事件预测器

接下来把所学的知识用于解决真实世界的问题。我们将建立一个SVM来预测一栋大楼进出楼门的人数。该数据集可以在 https://archive.ics.uci.edu/ml/datasets/CalIt2+Building+People+Counts 下载。我们将对数据集稍做调整，以便简化分析过程。调整过的数据集存放在building_event_binary.txt文件和building_event_multiclass.txt文件中。

3.7.1 准备工作

在建立模型之前，我们先看看数据格式。building_event_binary.txt文件的每一行都由6个逗号分割的字符串组成。这6个字符串的排序如下：

- 星期
- 日期
- 时间
- 离开大楼的人数
- 进入大楼的人数
- 是否有活动

前5个字符串组成输入数据，我们的任务是预测大楼是否举行活动。

building_event_multiclass.txt文件的每一行都由6个逗号分割的字符串组成。这个数据集比前面的更细，里面指明了大楼举行活动的类型。这6个字符串的排序如下：

- 星期
- 日期
- 时间
- 离开大楼的人数
- 进入大楼的人数
- 活动类型

前5个字符串是输入数据，我们的任务是预测大楼将举行什么活动。

3.7.2 详细步骤

(1) 读者可以参考event.py文件中的源代码。创建一个Python文件，然后加入下面的代码：

```
import numpy as np
from sklearn import preprocessing
from sklearn.svm import SVC
```

```
input_file = 'building_event_binary.txt'

# 读取数据
X = []
count = 0
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = line[:-1].split(',')
        X.append([data[0]] + data[2:])

X = np.array(X)
```

我们把数据全部加载到变量X中。

(2) 下面将字符串格式转换成数值格式：

```
# 将字符串转换成数值
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

(3) 用径向基函数、概率输出和类型平衡方法训练SVM分类器：

```
# 建立SVM模型
params = {'kernel': 'rbf', 'probability': True, 'class_weight': 'auto'}
classifier = SVC(**params)
classifier.fit(X, y)
```

(4) 现在就可以进行交叉验证了：

```
# 交叉验证
from sklearn import cross_validation

accuracy = cross_validation.cross_val_score(classifier,
        X, y, scoring='accuracy', cv=3)
print "Accuracy of the classifier: " + str(round(100*accuracy.mean(), 2)) + "%"
```

(5) 用一个新的数据点测试SVM：

```
# 对单一数据示例进行编码测试
input_data = ['Tuesday', '12:30:00','21','23']
input_data_encoded = [-1] * len(input_data)
count = 0
for i,item in enumerate(input_data):
```

```
        if item.isdigit():
            input_data_encoded[i] = int(input_data[i])
        else:
            input_data_encoded[i] = int(label_encoder[count].transform(input_data[i]))
            count = count + 1

input_data_encoded = np.array(input_data_encoded)

# 为特定数据点预测并打印输出结果
output_class = classifier.predict(input_data_encoded)
print "Output class:", label_encoder[-1].inverse_transform(output_class)[0]
```

(6) 运行代码，可以在命令行工具中看到如下结果：

```
Accuracy of the classifier: 89.88%
Output class: event
```

(7) 如果用building_event_multiclass.txt文件代替building_event_binary.txt文件作为输入数据文件，可以在命令行工具中看到以下结果：

```
Accuracy of the classifier: 65.9%
Output class: eventA
```

3.8 估算交通流量

根据相关数据预测交通流量是SVM的一个有趣应用。在上一节中我们将SVM作为一个分类器，下面将它作为一个回归器来估算交通流量。

3.8.1 准备工作

我们将要使用的数据集可以在https://archive.ics.uci.edu/ml/datasets/Dodgers+Loop+Sensor下载。这个数据集统计了洛杉矶道奇棒球队（Los Angeles Dodgers）进行主场比赛期间，体育场周边马路通过的车辆数量，存放在traffic_data.txt文件中。每一行都包含用逗号分隔的字符串格式，如下所示：

- 星期
- 时间
- 对手球队
- 棒球比赛是否正在继续
- 通行汽车的数量

3.8.2 详细步骤

(1) 下面看看如何建立SVM回归器。大家可以参考traffic.py文件中的源代码。创建一个Python

文件，然后加入下面的代码：

```
# 用SVM分类器估算交通流量

import numpy as np
from sklearn import preprocessing
from sklearn.svm import SVR

input_file = 'traffic_data.txt'

# 读取数据
X = []
count = 0
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = line[:-1].split(',')
        X.append(data)

X = np.array(X)
```

我们把数据全部加载到变量X中。

(2) 对数据进行编码：

```
# 将字符串转换成数值
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

(3) 用径向基函数创建并训练SVM回归器：

```
# 建立SVR
params = {'kernel': 'rbf', 'C': 10.0, 'epsilon': 0.2}
regressor = SVR(**params)
regressor.fit(X, y)
```

在上面的代码中，参数C指定了对错误分类的惩罚，参数epsilon指定了不使用惩罚的限制。

(4) 用交叉验证来检查回归器的性能：

```
# 交叉验证
import sklearn.metrics as sm

y_pred = regressor.predict(X)
print "Mean absolute error =", round(sm.mean_absolute_error(y, y_pred), 2)
```

(5) 在一个数据点上进行测试：

```
# 对单一数据示例进行编码测试
input_data = ['Tuesday', '13:35', 'San Francisco', 'yes']
input_data_encoded = [-1] * len(input_data)
count = 0
for i,item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] = int(label_encoder[count].transform(input_data[i]))
        count = count + 1

input_data_encoded = np.array(input_data_encoded)

# 为特定数据点预测并打印分类结果
print "Predicted traffic:", int(regressor.predict(input_data_encoded)[0])
```

(6) 运行代码，可以在命令行工具中看到以下结果：

```
Mean absolute error = 4.08
Predicted traffic: 29
```

第 4 章

无监督学习——聚类

4

在这一章，我们将介绍以下主题：

- 用k-means算法聚类数据
- 用向量量化（vector quantization）压缩图片
- 建立均值漂移（Mean Shift）聚类模型
- 用凝聚层次聚类（agglomerative clustering）进行数据分组
- 评价聚类算法的聚类效果
- 用DBSCAN算法自动估算集群数量
- 探索股票数据的模式
- 建立客户细分模型

4.1 简介

无监督学习是一种对不含标记的数据建立模型的机器学习范式。到目前为止，我们处理的数据都带有某种形式的标记，也就是说，学习算法可以根据标记看到这些数据，并对数据进行分类。但是，在无监督学习的世界中，我们没有这样的条件了。当需要用一些相似性指标对数据集进行分组时，就会用到这些算法了。

最常见的无监督学习方法就是**聚类**，你一定对这个词耳熟能详。当需要把无标记的数据分成几种集群时，就要用它来分析。这些集群通常是根据某种相似度指标进行划分的，例如欧氏距离（Euclidean distance）。无监督学习广泛应用于各种领域，如数据挖掘、医学影像、股票市场分析、计算机视觉、市场细分等。

4.2 用 k-means 算法聚类数据

k-means算法是最流行的聚类算法之一。这个算法常常利用数据的不同属性将输入数据划分为*k*组。分组是使用最优化的技术实现的，即让各组内的数据点与该组中心点的距离平方和最小化。如果你对k-means算法不太了解，可以在http://www.onmyphd.com/?p=k-means.clustering&

ckattempt=1上学习。

详细步骤

(1) 本节的完整代码已经放在kmeans.py文件中。我们先创建一个新的Python文件，然后导入下面的程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans

import utilities
```

(2) 加载输入数据，然后定义集群的数量。我们将使用data_multivar.txt 数据文件：

```
data = utilities.load_data('data_multivar.txt')
num_clusters = 4
```

(3) 我们需要看看输入数据是什么样子的。继续向Python文件中加入下面的代码：

```
plt.figure()
plt.scatter(data[:,0], data[:,1], marker='o',
        facecolors='none', edgecolors='k', s=30)
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
```

运行代码，可以看到如图4-1所示的图形。

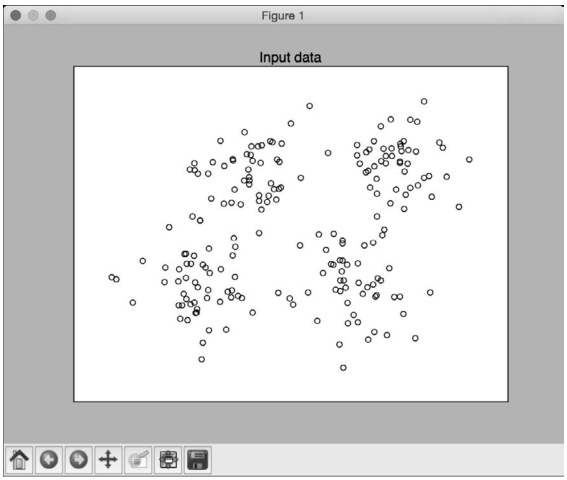


图　4-1

(4) 现在可以训练模型了。先初始化一个k-means对象，然后训练它：

```
kmeans = KMeans(init='k-means++', n_clusters=num_clusters, n_init=10)
kmeans.fit(data)
```

(5) 数据训练之后，我们需要可视化边界。继续向Python文件中加入下面的代码：

```
# 设置网格数据的步长
step_size = 0.01

# 画出边界
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
x_values, y_values = np.meshgrid(np.arange(x_min, x_max, step_size), np.arange(y_min,
y_max, step_size))

# 预测网格中所有数据点的标记
predicted_labels = kmeans.predict(np.c_[x_values.ravel(), y_values.ravel()])
```

(6) 我们已经通过网格数据评估了模型。接下来把这些结果都画出来，看看边界线的布局：

```
# 画出结果
predicted_labels = predicted_labels.reshape(x_values.shape)
plt.figure()
plt.clf()
plt.imshow(predicted_labels, interpolation='nearest',
           extent=(x_values.min(), x_values.max(), y_values.min(), y_values.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')

plt.scatter(data[:,0], data[:,1], marker='o',
        facecolors='none', edgecolors='k', s=30)
```

(7) 把中心点画在图形上：

```
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:,0], centroids[:,1], marker='o', s=200, linewidths=3,
        color='k', zorder=10, facecolors='black')
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Centoids and boundaries obtained using KMeans')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

运行代码，可以看到如图4-2所示的图形。

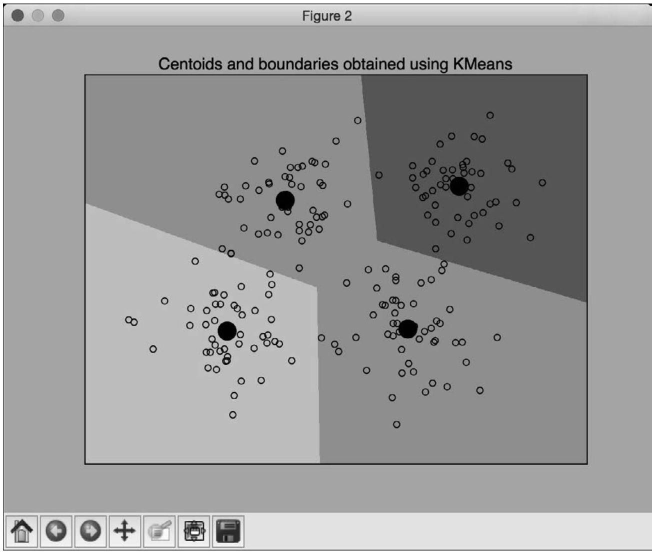


图　4-2

4.3　用矢量量化压缩图片

k-means聚类的主要应用之一就是**矢量量化**。简单来说，矢量量化就是"四舍五入"（rounding off）的*N*维版本。在处理数字等一维数据时，会用四舍五入技术减少存储空间。例如，如果只需要精确到两位小数，那么不会直接存储23.73473572，而是用23.73来代替。如果不关心小数部分，甚至可以直接存储24，这取决于我们的真实需求。

同理，当把四舍五入这个概念推广到*N*维数据时，就变成了矢量量化。当然，矢量量化的细节很多，你可以在http://www.data-compression.com/vq.shtml里学习更多的内容。矢量量化被广泛应用于图片压缩，我们用比原始图像更少的比特数来存储每个像素，从而实现图像图片。

详细步骤

(1) 本例的完整代码已经放在vector_quantization.py文件中。下面看看它是如何实现的。首先需要导入一些程序库。创建一个新的Python文件，然后加入下面的代码：

```
import argparse

import numpy as np
from scipy import misc
from sklearn import cluster
import matplotlib.pyplot as plt
```

(2) 创建一个函数，用来解析输入参数。我们需要把图片和每个像素被压缩的比特数传进去作为输入参数：

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Compress the input image \
            using clustering')
    parser.add_argument("--input-file", dest="input_file", required=True,
            help="Input image")
    parser.add_argument("--num-bits", dest="num_bits", required=False,
            type=int, help="Number of bits used to represent each pixel")
    return parser
```

(3) 再创建一个函数，用来压缩输入的图片：

```
def compress_image(img, num_clusters):
    # 将输入的图片转换成（样本量，特征量） 数组，以运行k-means聚类算法
    X = img.reshape((-1, 1))

    # 对输入数据运行k-means聚类
    kmeans = cluster.KMeans(n_clusters=num_clusters, n_init=4, random_state=5)
    kmeans.fit(X)
    centroids = kmeans.cluster_centers_.squeeze()
    labels = kmeans.labels_

    # 为每个数据配置离它最近的中心点，并转变为图片的形状
    input_image_compressed = np.choose(labels, centroids).reshape(img.shape)

    return input_image_compressed
```

(4) 压缩完图片之后，我们需要看看压缩算法对图片质量的影响。下面定义画图函数：

```
def plot_image(img, title):
    vmin = img.min()
    vmax = img.max()
    plt.figure()
    plt.title(title)
    plt.imshow(img, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)
```

(5) 我们现在已经准备好所有的函数了。下面定义主函数`main`，它可以把输入参数传进去并进行处理，然后提取输出图片：

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_file = args.input_file
    num_bits = args.num_bits

    if not 1 <= num_bits <= 8:
        raise TypeError('Number of bits should be between 1 and 8')

    num_clusters = np.power(2, num_bits)

    # 打印压缩率
```

```
compression_rate = round(100 * (8.0 - args.num_bits) / 8.0, 2)
print "\nThe size of the image will be reduced by a factor of", 8.0/args.num_bits
print "\nCompression rate = " + str(compression_rate) + "%"
```

(6) 加载输入图片：

```
# 加载输入图片
input_image = misc.imread(input_file, True).astype(np.uint8)

# 显示原始图片
plot_image(input_image, 'Original image')
```

(7) 用输入的参数压缩图片：

```
# 压缩图片
input_image_compressed = compress_image(input_image, num_clusters)
plot_image(input_image_compressed, 'Compressed image; compression rate = '
        + str(compression_rate) + '%')

plt.show()
```

(8) 现在我们的代码已经准备好了。在命令行工具中运行下面的命令：

```
$ python vector_quantization.py --input-file flower_image.jpg --num-bits 4
```

输入的图片如图4-3所示。

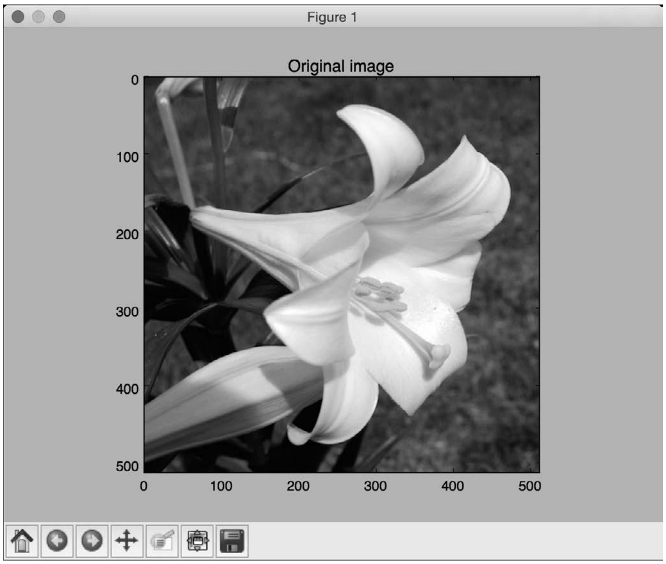


图　4-3

压缩过的图片如图4-4所示。

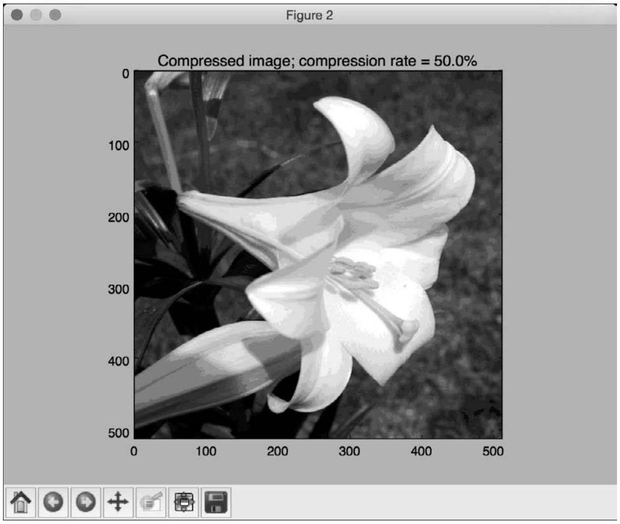


图 4-4

(9) 我们把每个像素的压缩比特数降到2，再压缩图片。在命令行工具中运行下面的命令：

```
$ python vector_quantization.py --input-file flower_image.jpg --num-bits 2
```

可以看到压缩过的图片如图4-5所示。

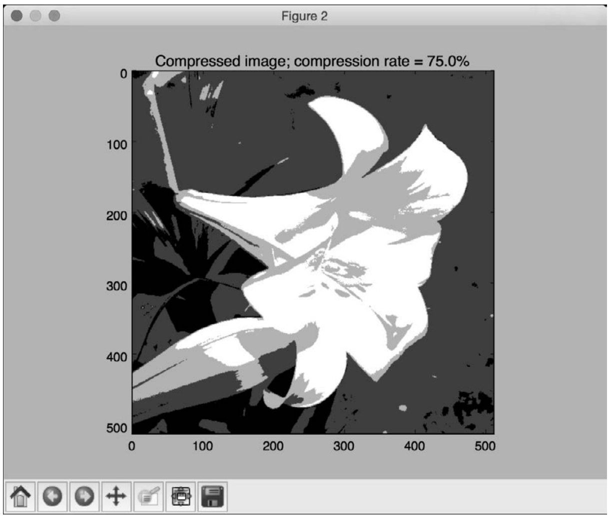


图 4-5

(10) 如果把每个像素的压缩比特数降到1，可以看到只有黑白两种颜色的二进制图像。运行下面的命令：

```
$ python vector_quantization.py --input-file flower_image.jpg --num-bits 1
```

图片压缩效果如图4-6所示。

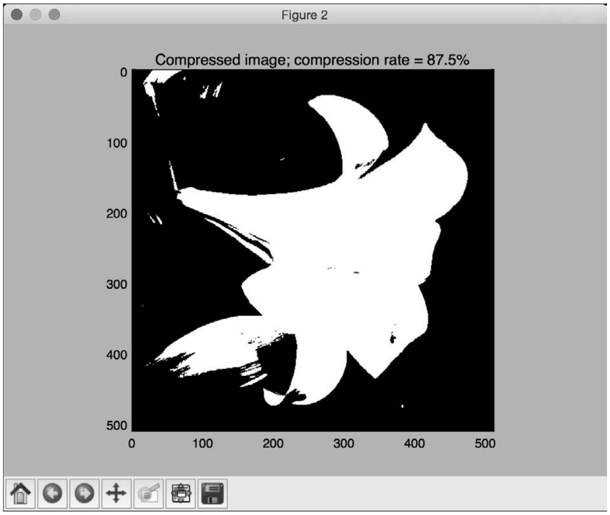


图　4-6

4.4　建立均值漂移聚类模型

均值漂移是一种非常强大的无监督学习算法，用于集群数据点。该算法把数据点的分布看成是概率密度函数（probability-density function），希望在特征空间中根据函数分布特征找出数据点的“模式”（mode）。这些“模式”就对应于一群群局部最密集（local maxima）分布的点。均值漂移算法的优点是它不需要事先确定集群的数量。

假设有一组输入点，我们要在不知道要寻找多少集群的情况下找到它们。均值漂移算法就可以把这些点看成是服从某个概率密度函数的样本。如果这些数据点有集群，那么它们对应于概率密度函数的峰值。该算法从一个随机点开始，逐渐收敛于各个峰值。你可以在 http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TUZEL1/MeanShift.pdf中学习更详细的内容。

详细步骤

(1) 本例的完整代码已经放在mean_shift.py文件中。我们看看它是如何实现的。首先创建一个新的Python文件，然后导入一些需要用到的程序包：

```
import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth

import utilities
```

(2) 从data_multivar.txt文件中加载输入数据：

```
# 从输入文件加载数据
X = utilities.load_data('data_multivar.txt')
```

(3) 通过指定输入参数创建一个均值漂移模型：

```
# 设置带宽参数bandwidth
bandwidth = estimate_bandwidth(X, quantile=0.1, n_samples=len(X))

# 用MeanShift计算聚类
meanshift_estimator = MeanShift(bandwidth=bandwidth, bin_seeding=True)
```

(4) 训练模型：

```
meanshift_estimator.fit(X)
```

(5) 提取标记：

```
labels = meanshift_estimator.labels_
```

(6) 从模型中提取集群的中心点，然后打印集群数量：

```
centroids = meanshift_estimator.cluster_centers_
num_clusters = len(np.unique(labels))

print "Number of clusters in input data =", num_clusters
```

(7) 把集群可视化：

```
# 画出数据点和聚类中心
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure()

# 为每种集群设置不同的标记
markers = '.*xv'
```

(8) 迭代数据点并画出它们：

```
for i, marker in zip(range(num_clusters), markers):
    # 画出属于某个集群中心的数据点
    plt.scatter(X[labels==i, 0], X[labels==i, 1], marker=marker, color='k')

    # 画出集群中心
    centroid = centroids[i]
    plt.plot(centroid[0], centroid[1], marker='o', markerfacecolor='k',
             markeredgecolor='k', markersize=15)

plt.title('Clusters and their centroids')
plt.show()
```

(9) 运行代码，可以看到如图4-7所示的图形。

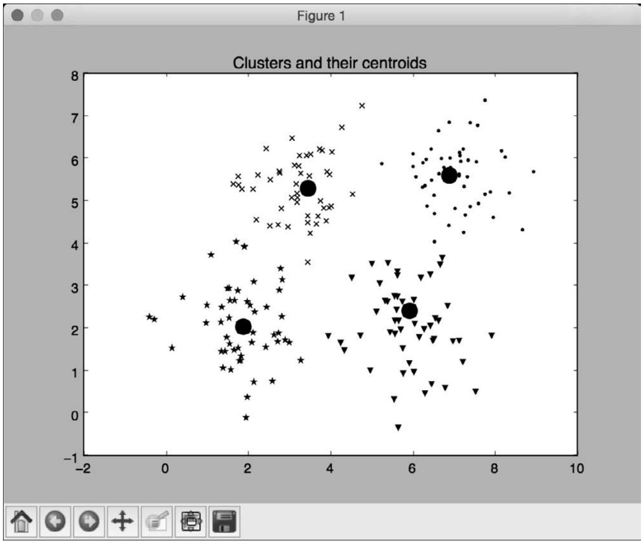


图　4-7

4.5　用凝聚层次聚类进行数据分组

在介绍凝聚层次聚类之前，我们需要先理解层次聚类（hierarchical clustering）。**层次聚类**是一组聚类算法，通过不断地分解或合并集群来构建树状集群（tree-like clusters）。层次聚类的结构可以用一颗树表示。

层次聚类算法可以是自下而上的，也可以是自上而下的。具体是什么含义呢？在自下而上的算法中，每个数据点都被看作是一个单独的集群。这些集群不断地合并，直到所有的集群都合并成一个巨型集群。这被称为**凝聚层次聚类**。与之相反的是，自上而下层次的算法是从一个巨大的集群开始，不断地分解，直到所有的集群变成一个单独的数据点。你可以在http://nlp.stanford.edu/IR-book/html/htmledition/hierarchical-agglomerative-clustering-1.html学习更多的内容。

详细步骤

(1) 本例的完整代码都已经放在agglomerative.py文件中。让我们看看它是如何实现的。首先创建一个新的Python文件，然后导入一些需要用到的程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from sklearn.neighbors import kneighbors_graph
```

(2) 定义一个实现凝聚层次聚类的函数：

```
def perform_clustering(X, connectivity, title, num_clusters=3, linkage='ward'):
    plt.figure()
    model = AgglomerativeClustering(linkage=linkage,
                    connectivity=connectivity, n_clusters=num_clusters)
    model.fit(X)
```

(3) 提取标记，然后指定不同聚类在图形中的标记：

```
    # 提取标记
    labels = model.labels_

    # 为每种集群设置不同的标记
    markers = '.vx'
```

(4) 迭代数据，用不同的标记把聚类的点画在图形中：

```
    for i, marker in zip(range(num_clusters), markers):
        # 画出属于某个集群中心的数据点
        plt.scatter(X[labels==i, 0], X[labels==i, 1], s=50,
                    marker=marker, color='k', facecolors='none')

    plt.title(title)
```

(5) 为了演示凝聚层次聚类的优势，我们用它对一些在空间中是连接在一起、但彼此却非常接近的数据进行聚类。我们希望连接在一起的数据可以聚成一类，而不是在空间上非常接近的点聚成一类。下面定义一个函数来获取一组呈螺旋状的数据点：

```
def get_spiral(t, noise_amplitude=0.5):
    r = t
    x = r * np.cos(t)
    y = r * np.sin(t)

    return add_noise(x, y, noise_amplitude)
```

(6) 在上面的函数中，我们增加了一些噪声，因为这样做可以增加一些不确定性。下面定义噪声函数：

```
def add_noise(x, y, amplitude):
    X = np.concatenate((x, y))
    X += amplitude * np.random.randn(2, X.shape[1])
    return X.T
```

(7) 我们再定义一个函数来获取位于玫瑰曲线上的数据点（rose curve，又称为rhodonea curve，极坐标中的正弦曲线）：

```
def get_rose(t, noise_amplitude=0.02):
    # 设置玫瑰曲线方程；如果变量k是奇数，那么曲线有k朵花瓣；如果k是偶数，那么有2k朵花瓣
    k = 5
    r = np.cos(k*t) + 0.25
    x = r * np.cos(t)
    y = r * np.sin(t)

    return add_noise(x, y, noise_amplitude)
```

(8) 为了增加多样性，我们再定义一个hypotrochoid函数：

```
def get_hypotrochoid(t, noise_amplitude=0):
    a, b, h = 10.0, 2.0, 4.0
    x = (a - b) * np.cos(t) + h * np.cos((a - b) / b * t)
    y = (a - b) * np.sin(t) - h * np.sin((a - b) / b * t)

    return add_noise(x, y, 0)
```

(9) 现在可以定义主函数main了：

```
if __name__=='__main__':
    # 生成样本数据
    n_samples = 500
    np.random.seed(2)
    t = 2.5 * np.pi * (1 + 2 * np.random.rand(1, n_samples))
    X = get_spiral(t)

    # 不考虑螺旋形的数据连接性
    connectivity = None
    perform_clustering(X, connectivity, 'No connectivity')

    # 根据数据连接线创建K个临近点的图形
    connectivity = kneighbors_graph(X, 10, include_self=False)
    perform_clustering(X, connectivity, 'K-Neighbors connectivity')

    plt.show()
```

(10) 运行代码，可以看到如图4-8所示的图形（没有用任何连接特征）。

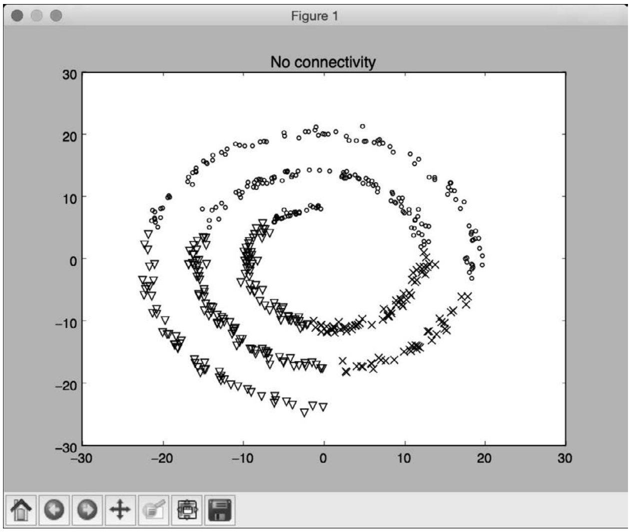


图　4-8

(11) 还可以看到如图4-9所示的图形（使用连接特征）。

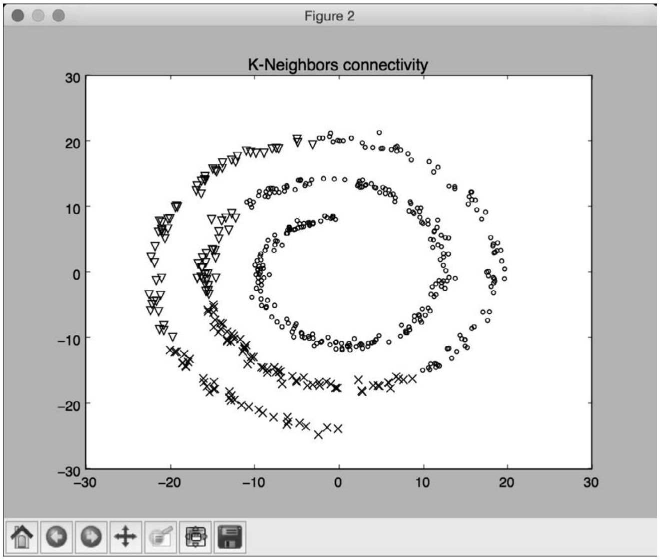


图 4-9

从图4-8和图4-9中可以看出，使用连接特征可以让我们把连接在一起的数据合成一组，而不是按照它们在螺旋线上的位置进行聚类。

4.6 评价聚类算法的聚类效果

到目前为止，我们已经介绍了3种聚类算法，却没有度量过它们的聚类效果。在监督学习中，可以用预测值与原始值进行比较来计算模型的准确性，但是，在无监督学习中，我们的数据没有标记，因此，需要一种度量聚类算法的方法。

度量聚类算法的一个好方法是观察集群被分离的离散程度。这些集群是不是被分离得很合理？一个集群中所有的数据点是不是足够紧密？需要拟定一个指标来衡量这种特征，于是，我们采用一个被称为**轮廓系数**（Silhouette Coefficient）得分的指标。该得分是为每个数据点定义的，它的定义如下：

$$得分 = (x - y) / \max(x, y)$$

其中，x表示在同一个集群中某个数据点与其他数据点的平均距离，y表示某个数据点与最近的另一个集群的所有点的平均距离。

详细步骤

(1) 本例的完整代码已经放在performance.py文件中。我们看看它是如何实现的。首先创建一

个新的Python文件，然后导入一些需要用到的程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans

import utilities
```

(2) 从data_perf.txt文件中加载输入数据：

```
# 加载数据
data = utilities.load_data('data_perf.txt')
```

(3) 为了确定集群的最佳数量，我们迭代一系列的值，找出其中的峰值：

```
scores = []
range_values = np.arange(2, 10)

for i in range_values:
    # 训练模型
    kmeans = KMeans(init='k-means++', n_clusters=i, n_init=10)
    kmeans.fit(data)
    score = metrics.silhouette_score(data, kmeans.labels_,
                metric='euclidean', sample_size=len(data))

    print "\nNumber of clusters =", i
    print "Silhouette score =", score

    scores.append(score)
```

(4) 画出图形并找出峰值：

```
# 画出得分条形图
plt.figure()
plt.bar(range_values, scores, width=0.6, color='k', align='center')
plt.title('Silhouette score vs number of clusters')

# 画出数据
plt.figure()
plt.scatter(data[:,0], data[:,1], color='k', s=30, marker='o', facecolors='none')
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

plt.show()
```

(5) 运行代码，可以在命令行工具中看到如图4-10所示的结果。

```
Number of clusters = 2
Silhouette score = 0.529039717547

Number of clusters = 3
Silhouette score = 0.557246639118

Number of clusters = 4
Silhouette score = 0.583275751783

Number of clusters = 5
Silhouette score = 0.658279690976

Number of clusters = 6
Silhouette score = 0.582358411948

Number of clusters = 7
Silhouette score = 0.528610740989

Number of clusters = 8
Silhouette score = 0.459759448983

Number of clusters = 9
Silhouette score = 0.415953573837
```

图 4-10

(6) 画出的条形图如图4-11所示。

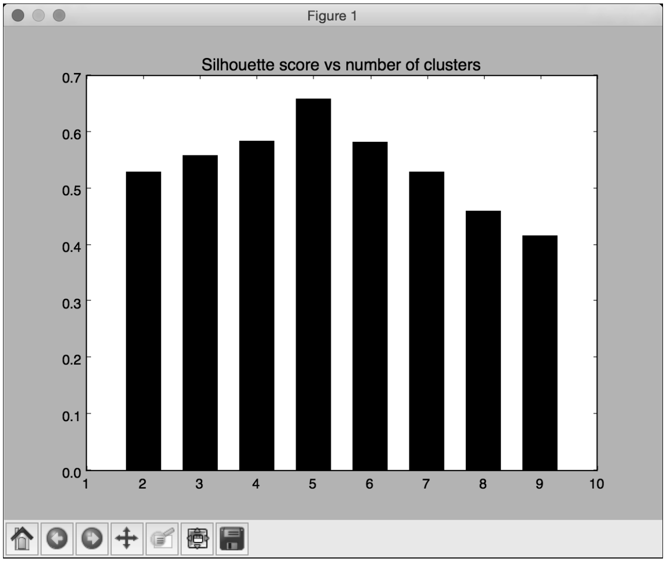


图 4-11

(7) 从图4-11中可以看出，5个集群是最好的配置，此时的实际图形如图4-12所示。

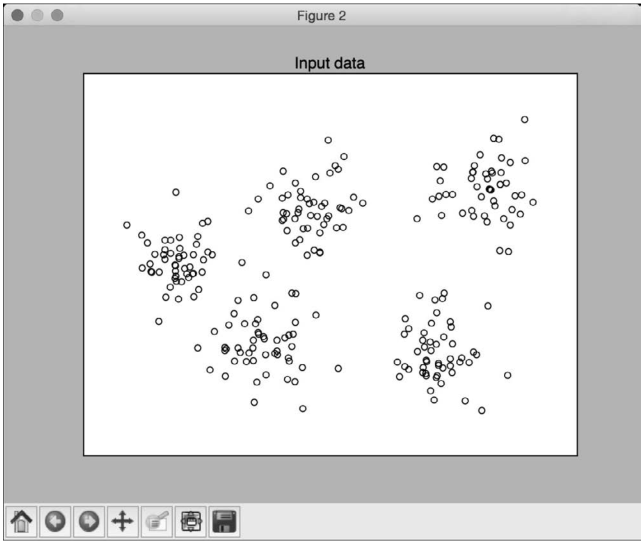


图　4-12

可以从图4-12中直观地确认数据实际上有5个集群。我们只是采用了一个包含5个不同集群的小数据集的例子。通过轮廓系数判断聚类效果的方法，对那些包含不容易可视化的高维数据的大型数据集非常有用。

4.7　用 DBSCAN 算法自动估算集群数量

介绍k-means算法的时候，必须把集群数量当作一个输入参数。在真实世界中，我们事先并不知道这个信息。可以搜索集群数量的参数空间，通过轮廓系数得分找到最优的集群数量，但这是一个非常耗时的过程。难道就没有一种方法可以直接找出集群数量吗？DBSCAN（Density-Based Spatial Clustering of Applications with Noise，带噪声的基于密度的聚类方法）应运而生。

DBSCAN将数据点看成是紧密集群的若干组。如果某个点属于一个集群，那么就应该有许多点也属于同一个集群。该方法里面有一个`epsilon`参数，可以控制这个点到其他点的最大距离。如果两个点的距离超过了参数`epsilon`的值，它们就不可能在一个集群中。你可以在http://staffwww.itn.liu.se/~aidvi/courses/06/dm/Seminars2011/DBSCAN(4).pdf学习更多的内容。这种方法的主要优点是它可以处理异常点。如果有一些点位于数据稀疏区域，DBSCAN就会把这些点作为异常点，而不会强制将它们放入一个集群中。

详细步骤

(1) 本例的完整代码已经放在estimate_clusters.py文件中。我们看看它是如何实现的。首先创

建一个新的Python文件，然后导入一些需要用到的程序库：

```
from itertools import cycle

import numpy as np
from sklearn.cluster import DBSCAN
from sklearn import metrics
import matplotlib.pyplot as plt

from utilities import load_data2
```

(2) 从data_perf.txt文件中加载输入数据。这和上一例的数据文件一样，这样可以帮助我们用同样的数据集对比两种方法：

```
# 加载输入数据
input_file = 'data_perf.txt'
X = load_data(input_file)
```

(3) 我们需要找到最佳集群数量参数。先初始化一些变量：

```
# 寻找最优的epsilon参数值
eps_grid = np.linspace(0.3, 1.2, num=10)
silhouette_scores = []
eps_best = eps_grid[0]
silhouette_score_max = -1
model_best = None
labels_best = None
```

(4) 搜索参数空间：

```
for eps in eps_grid:
    # 训练DBSCAN聚类模型
    model = DBSCAN(eps=eps, min_samples=5).fit(X)

    # 提取标记
    labels = model.labels_
```

(5) 每次迭代，我们都需要提取性能指标：

```
    # 提取性能指标
    silhouette_score = round(metrics.silhouette_score(X, labels), 4)
    silhouette_scores.append(silhouette_score)

    print "Epsilon:", eps, " --> silhouette score:", silhouette_score
```

(6) 我们需要保存指标的最佳得分和对应的epsilon值：

```
    if silhouette_score > silhouette_score_max:
        silhouette_score_max = silhouette_score
        eps_best = eps
        model_best = model
        labels_best = labels
```

(7) 画出条形图：

```
# 画出条形图
plt.figure()
plt.bar(eps_grid, silhouette_scores, width=0.05, color='k', align='center')
plt.title('Silhouette score vs epsilon')

# 打印最优参数
print "\nBest epsilon =", eps_best
```

(8) 把最优的模型和标记保存起来：

```
# 最优参数对应的模型与标记
model = model_best
labels = labels_best
```

(9) 有些数据点还没有分配集群，我们需要识别它们：

```
# 检查标记中没有分配集群的数据点
offset = 0
if -1 in labels:
    offset = 1
```

(10) 提取集群的数量：

```
# 数据中的集群数量
num_clusters = len(set(labels)) - offset

print "\nEstimated number of clusters =", num_clusters
```

(11) 提取核心样本：

```
# 从训练模型中提取核心样本的数据点索引
mask_core = np.zeros(labels.shape, dtype=np.bool)
mask_core[model.core_sample_indices_] = True
```

(12) 接下来将集群结果可视化。首先提取独特的标记集合，然后分配不同的标记：

```
# 画出集群结果
plt.figure()
labels_uniq = set(labels)
markers = cycle('vo^s<>')
```

(13) 用迭代法把每个集群的数据点用不同的标记画出来：

```
for cur_label, marker in zip(labels_uniq, markers):
    # 用黑点表示未分配的数据点
    if cur_label == -1:
        marker = '.'

    # 为当前标记添加符号
    cur_mask = (labels == cur_label)
```

```
    cur_data = X[cur_mask & mask_core]
    plt.scatter(cur_data[:, 0], cur_data[:, 1], marker=marker,
             edgecolors='black', s=96, facecolors='none')

    cur_data = X[cur_mask & ~mask_core]
    plt.scatter(cur_data[:, 0], cur_data[:, 1], marker=marker,
             edgecolors='black', s=32)

plt.title('Data separated into clusters')
plt.show()
```

(14) 运行代码，可以在命令行工具中看到如图4-13所示的结果。


```
Epsilon: 0.3  --> silhouette score: 0.1287
Epsilon: 0.4  --> silhouette score: 0.3594
Epsilon: 0.5  --> silhouette score: 0.5134
Epsilon: 0.6  --> silhouette score: 0.6165
Epsilon: 0.7  --> silhouette score: 0.6322
Epsilon: 0.8  --> silhouette score: 0.6366
Epsilon: 0.9  --> silhouette score: 0.5142
Epsilon: 1.0  --> silhouette score: 0.5629
Epsilon: 1.1  --> silhouette score: 0.5629
Epsilon: 1.2  --> silhouette score: 0.5629

Best epsilon = 0.8

Estimated number of clusters = 5
```

图　4-13

(15) 条形图如图4-14所示。

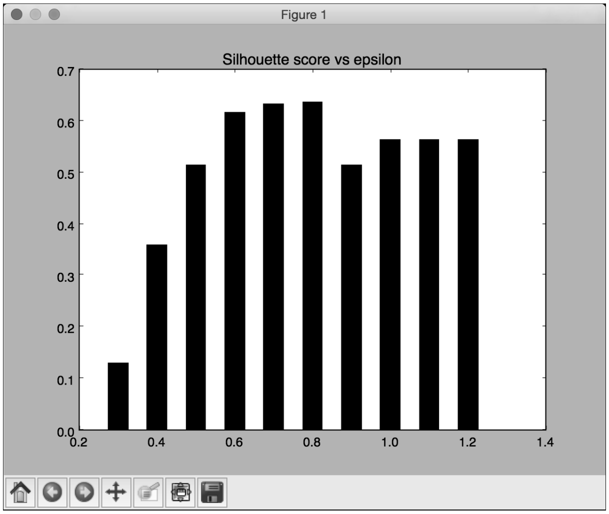


图　4-14

(16) 用实心标注的未被分配的数据点如图4-15所示。

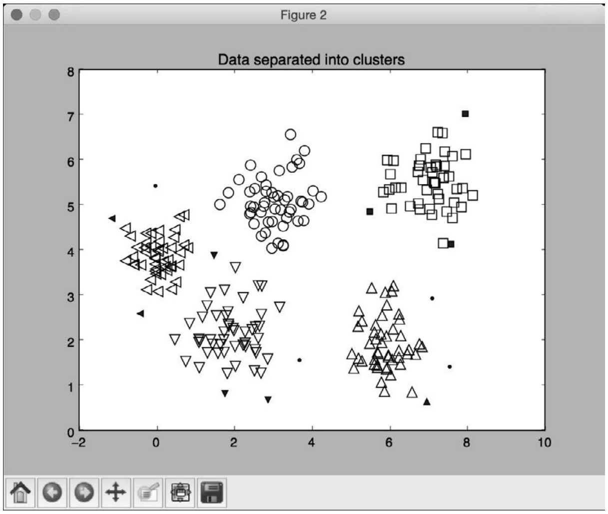


图　4-15

4.8　探索股票数据的模式

让我们看看如何用无监督学习进行股票数据分析。假设我们并不知道股票市场有多少集群，因此需要用一种**近邻传播聚类**（Affinity Propagation）算法来集群。这种算法会找出数据中每个集群的代表性数据点，会找到数据点间的相似性度量值，并把所有数据点看成潜在的代表性数据点，也称为**取样器**（exemplar）。更多内容可参考 http://www.cs.columbia.edu/~delbert/docs/DDueck-thesis_small.pdf。

本例将分析在特定时间内的股票市场变化，我们的目标是根据股价的波动找出公司行为的相似性。

详细步骤

(1) 本例的完整代码已经放在stock_market.py文件中。我们看看它是如何实现的。首先创建一个新的Python文件，然后导入一些需要用到的程序包：

```
import json
import datetime

import numpy as np
import matplotlib.pyplot as plt
from sklearn import covariance, cluster
from matplotlib.finance import quotes_historical_yahoo_ochl as quotes_yahoo
```

(2) 我们需要一个包含所有符号以及对应名称的文件，具体信息在symbol_map.json文件中。下面加载这个文件：

```
# 输入符号信息文件
symbol_file = 'symbol_map.json'
```

(3) 从符号映射文件中读取数据：

```
# 加载符号映射信息
with open(symbol_file, 'r') as f:
    symbol_dict = json.loads(f.read())

symbols, names = np.array(list(symbol_dict.items())).T
```

(4) 让我们指定分析的时间段。将用这个时间段作为输入数据的起止时间：

```
# 选择时间段
start_date = datetime.datetime(2004, 4, 5)
end_date = datetime.datetime(2007, 6, 2)
```

(5) 读取输入的数据：

```
quotes = [quotes_yahoo(symbol, start_date, end_date, asobject=True)
                for symbol in symbols]
```

(6) 由于需要分析一些特征点，我们使用每天的开盘价和收盘价的差异来分析数据：

```
# 提取开盘价和收盘价
opening_quotes = np.array([quote.open for quote in quotes]).astype(np.float)
closing_quotes = np.array([quote.close for quote in quotes]).astype(np.float)

# 计算每日股价波动（收盘价-开盘价）
delta_quotes = closing_quotes - opening_quotes
```

(7) 建立一个协方差图模型：

```
# 从相关性中建立协方差图模型
edge_model = covariance.GraphLassoCV()
```

(8) 在使用数据之前先对它进行标准化：

```
# 数据标准化
X = delta_quotes.copy().T
X /= X.std(axis=0)
```

(9) 用数据训练模型：

```
# 训练模型
with np.errstate(invalid='ignore'):
    edge_model.fit(X)
```

(10) 我们现在已经准备好建立聚类模型了：

```
# 用近邻传播算法建立聚类模型
_, labels = cluster.affinity_propagation(edge_model.covariance_)
```

```
num_labels = labels.max()

# 打印聚类结果
for i in range(num_labels + 1):
    print "Cluster", i+1, "-->", ', '.join(names[labels == i])
```

运行代码，可以在命令行工具中看到如图4-16所示的结果。

```
Cluster 1 --> ConocoPhillips, Chevron, Total, Valero Energy, Exxon
Cluster 2 --> CVS, Walgreen
Cluster 3 --> IBM, Cisco, Microsoft, Texas instruments, Ford, HP, Dell
Cluster 4 --> Cablevision
Cluster 5 --> Pfizer, Apple, Caterpillar, Canon, Boeing, Toyota, SAP, Honda, Mitsubishi, Sony, Mc Donalds
, Unilever, Wal-Mart
Cluster 6 --> Kimberly-Clark, Colgate-Palmolive, Procter Gamble
Cluster 7 --> Yahoo, Amazon
Cluster 8 --> American express, Wells Fargo, Navistar, Bank of America, Time Warner, Ryder, Kellogg, Home
 Depot, AIG, Goldman Sachs, General Electrics, Marriott, Xerox, JPMorgan Chase, DuPont de Nemours, 3M, Co
mcast
Cluster 9 --> GlaxoSmithKline, Novartis, Sanofi-Aventis
Cluster 10 --> Pepsi, Coca Cola
Cluster 11 --> Raytheon, Lookheed Martin, General Dynamics, Northrop Grumman
Cluster 12 --> Kraft Foods
```

图　4-16

4.9　建立客户细分模型

无监督学习的主要应用场景之一就是市场细分。虽然在我们开发市场时获取的数据都没有标记，但是将市场细分成不同类型至关重要，这样人们就可以关注各自的市场类型了。市场细分对广告投放、库存管理、配送策略的实施、大众传媒等市场行为都非常有用。下面把无监督学习应用到一个市场细分的案例上，看看效果如何。

我们将与一个零售商和他的客户打交道，采用https://archive.ics.uci.edu/ml/datasets/Wholesale+customers的数据进行分析。数据表里包含了不同类型商品的销售数据，目标是找到数据集群，从而为客户提供最优的销售和分销策略。

详细步骤

(1) 本例的完整代码已经放在customer_segmentation.py文件中。我们看看它是如何实现的。首先创建一个新的Python文件，然后导入一些需要用到的程序包：

```
import csv

import numpy as np
from sklearn import cluster, covariance, manifold
from sklearn.cluster import MeanShift, estimate_bandwidth
import matplotlib.pyplot as plt
```

(2) 从wholesale.csv文件中加载输入数据：

```
# 从输入文件加载数据
input_file = 'wholesale.csv'
file_reader = csv.reader(open(input_file, 'rb'), delimiter=',')
X = []
for count, row in enumerate(file_reader):
    if not count:
        names = row[2:]
        continue

    X.append([float(x) for x in row[2:]])

# 转换为numpy数组
X = np.array(X)
```


(3) 和前面的内容一样，建立一个均值漂移聚类模型：

```
# 估计带宽参数bandwidth
bandwidth = estimate_bandwidth(X, quantile=0.8, n_samples=len(X))

# 用MeanShift函数计算聚类
meanshift_estimator = MeanShift(bandwidth=bandwidth, bin_seeding=True)
meanshift_estimator.fit(X)
labels = meanshift_estimator.labels_
centroids = meanshift_estimator.cluster_centers_
num_clusters = len(np.unique(labels))

print "\nNumber of clusters in input data =", num_clusters
```

(4) 打印获得的集群中心：

```
print "\nCentroids of clusters:"
print '\t'.join([name[:3] for name in names])
for centroid in centroids:
    print '\t'.join([str(int(x)) for x in centroid])
```

(5) 把两个特征（milk和groceries）的聚类结果可视化，以获取直观的输出：

```
# 数据可视化

centroids_milk_groceries = centroids[:, 1:3]

# 用centroids_milk_groceries中的坐标画出中心点
plt.figure()
plt.scatter(centroids_milk_groceries[:,0], centroids_milk_groceries[:,1],
        s=100, edgecolors='k', facecolors='none')

offset = 0.2
plt.xlim(centroids_milk_groceries[:,0].min() - offset *
centroids_milk_groceries[:,0].ptp(),
        centroids_milk_groceries[:,0].max() + offset *
centroids_milk_groceries[:,0].ptp(),)
plt.ylim(centroids_milk_groceries[:,1].min() - offset *
centroids_milk_groceries[:,1].ptp(),
```

```
        centroids_milk_groceries[:,1].max() + offset *
centroids_milk_groceries[:,1].ptp())

plt.title('Centroids of clusters for milk and groceries')
plt.show()
```

(6) 运行代码，可以在命令行工具中看到如图4-17所示的结果。

```
Number of clusters in input data = 8

Centroids of clusters:
Fre     Mil     Gro     Fro     Det     Del
9632    4671    6593    2570    2296    1248
40204   46314   57584   5518    25436   4241
8565    4980    67298   131     38102   1215
32717   16784   13626   60869   1272    5609
22925   73498   32114   987     20070   903
112151  29627   18148   16745   4948    8550
16117   46197   92780   1026    40827   2944
36847   43950   20170   36534   239     47943
```

图　4-17

(7) 如图4-18所示描绘的是milk（牛奶）和groceries（杂货）两个特征的聚类中心，其中milk特征值对应X轴坐标，groceries特征值对应Y轴坐标。

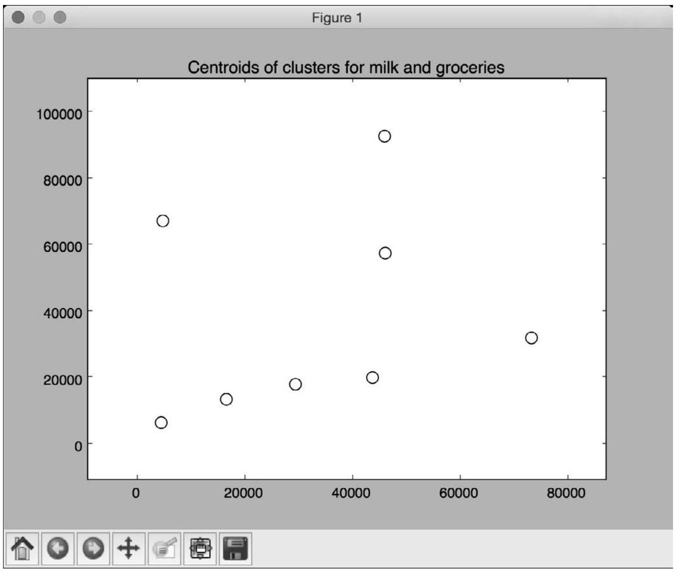


图　4-18

第 5 章

构建推荐引擎

5

在这一章，我们将介绍以下主题：

- 为数据处理构建函数组合
- 构建机器学习流水线（pipeline）
- 寻找最近邻
- 构建一个KNN分类器
- 构建一个KNN回归器
- 计算欧氏距离分数（Euclidean distance score）
- 计算皮尔逊相关系数（Pearson correlation score）
- 寻找数据集中的相似用户
- 生成电影推荐

5.1 简介

推荐引擎是一个能预测用户兴趣点的模型。将推荐引擎应用于电影语境时，便成了一个电影推荐引擎。我们通过预测当前用户可能会喜欢的内容，将相应的东西从数据库中筛选出来，这样的推荐引擎可以有助于将用户和数据集中的合适内容连接起来。为什么推荐引擎这么重要？设想你有一个很庞大的商品目录，而用户可能或者不可能查找所有的相关内容。通过推荐合适的内容，可以增加用户消费。有些公司（如Netflix）严重地依赖推荐系统来保持用户参与度。

推荐引擎通常用协同过滤（collaborative filtering）或基于内容的过滤（content-based filtering）来产生一组推荐。两种过滤方法的不同之处在于挖掘推荐的方式。协同过滤从当前用户过去的行为和其他用户对当前用户的评分来构建模型，然后使用这个模型来预测这个用户可能感兴趣的内容；而基于内容的过滤用商品本身的特征来给用户推荐更多的商品，商品间的相似度是模型主要的关注点。本章将重点介绍协同过滤。

5.2　为数据处理构建函数组合

机器学习系统中的主要组成部分是数据处理流水线。在数据被输入到机器学习算法中进行训练之前，需要对数据做各种方式的处理，使得该数据可以被算法利用。在构建一个准确的、可扩展的机器学习系统的过程中，拥有一个健壮的数据处理流水线非常重要。有很多基本的函数功能可以使用，通常数据处理流水线就是这些基本函数的组合。不推荐使用嵌套或循环的方式调用这些函数，而是用函数式编程的方式构建函数组合。接下来介绍如何组合这些函数来形成一个可重用的函数组合，本节将创建3个基本函数，并介绍如何将其组合成一个流水线。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import numpy as np
```

(2) 定义第一个函数，将数组的每一个元素加3：

```
def add3(input_array):
    return map(lambda x: x+3, input_array)
```

(3) 定义第二个函数，将数组的每一个元素乘以2：

```
def mul2(input_array):
    return map(lambda x: x*2, input_array)
```

(4) 定义第三个函数，将数组的每一个元素减去5：

```
def sub5(input_array):
    return map(lambda x: x-5, input_array)
```

(5) 定义一个函数组合器，将这些函数作为输入参数，返回一个组合函数。这个组合函数基本上是输入函数按序执行的一个函数：

```
def function_composer(*args):
    return reduce(lambda f, g: lambda x: f(g(x)), args)
```

我们用reduce函数依次执行所有函数，也就是将所有的输入函数合并。

(6) 接下来可以做函数组合了。首先定义一些数据和一组操作：

```
if __name__=='__main__':
    arr = np.array([2,5,4,7])

    print "\nOperation: add3(mul2(sub5(arr)))"
```

(7) 如果用常规的方法，我们依次执行函数，代码如下：

```
    arr1 = add3(arr)
```

```
arr2 = mul2(arr1)
arr3 = sub5(arr2)
print "Output using the lengthy way:", arr3
```

(8) 下面用单行代码的函数组合器实现同样的功能：

```
func_composed = function_composer(sub5, mul2, add3)
print "Output using function composition:", func_composed(arr)
```

(9) 可以通过上面的方法用单行代码实现函数组合，但是其表示方式是嵌套的和不可读的，而且也是不可重用的。当需要再次用到这组操作时，需要重新编写：

```
print "\nOperation: sub5(add3(mul2(sub5(mul2(arr)))))\nOutput:", \
        function_composer(mul2, sub5, mul2, add3, sub5)(arr)
```

(10) 运行代码，可以在命令行工具中看到如图5-1所示的输出结果。

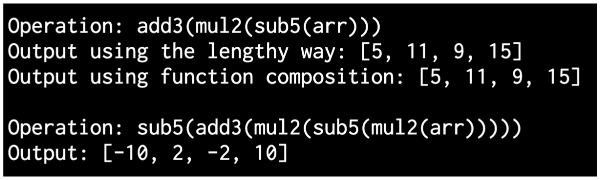


图　5-1

5.3　构建机器学习流水线

scikit-learn库中包含了构建机器学习流水线的方法。只需要指定函数，它就会构建一个组合对象，使数据通过整个流水线。这个流水线可以包括诸如预处理、特征选择、监督式学习、非监督式学习等函数。这一节将构建一个流水线，以便输入特征向量、选择最好的*k*个特征、用随机森林分类器进行分类等。

5.3.1　详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
from sklearn.datasets import samples_generator
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
```

(2) 生成一些示例数据：

```
# 生成样本数据
X, y = samples_generator.make_classification(
        n_informative=4, n_features=20, n_redundant=0, random_state=5)
```

这一行代码生成了一个20维的特征向量，因为这是一个默认值。在这里，你可以通过修改n_features参数来修改特征向量的维数。

(3) 建立流水线的第一步是在数据点进一步操作之前选择*k*个最好的特征。这里设置k的值为10：

```
# 特征选择器
selector_k_best = SelectKBest(f_regression, k=10)
```

(4) 建立流水线的第二步是用随机森林分类器分类数据：

```
# 随机森林分类器
classifier = RandomForestClassifier(n_estimators=50, max_depth=4)
```

(5) 接下来可以创建流水线了。pipeline方法允许我们用预定义的对象来创建流水线：

```
# 构建机器学习流水线
pipeline_classifier = Pipeline([('selector', selector_k_best), ('rf', classifier)])
```

还可以在流水线中为模块指定名称。上一行代码中将特征选择器命名为selector，将随机森林分类器命名为rf。你也可以任意选用其他名称。

(6) 也可以更新这些参数，用上一步骤中命名的名称设置这些参数。例如，如果希望在特征选择器中将k值设置为6，在随机森林分类器中将n_estimators的值设置为25，可以用下面的代码实现。注意，这些变量名称已在上一步骤中给出。

```
pipeline_classifier.set_params(selector__k=6,  rf__n_estimators=25)
```

(7) 接下来训练分类器：

```
# 训练分类器
pipeline_classifier.fit(X, y)
```

(8) 为训练数据预测输出结果：

```
# 预测输出结果
prediction = pipeline_classifier.predict(X)
print "\nPredictions:\n", prediction
```

(9) 评价分类器的性能：

```
# 打印分类器得分
print "\nScore:", pipeline_classifier.score(X, y)
```

(10) 还可以查看哪些特征被选中，并将其打印出：

```
# 打印被分类器选中的特征
features_status = pipeline_classifier.named_steps['selector'].get_support()
selected_features = []
for count, item in enumerate(features_status):
```

```
    if item:
        selected_features.append(count)

print "\nSelected features (0-indexed):", ', '.join([str(x) for x in
selected_features])
```

(11) 运行代码，可以在命令行工具中看到如图5-2所示的输出结果。

```
Predictions:
[0 0 0 1 1 0 1 1 0 1 0 0 0 0 1 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0
 0 0 0 0 0 0 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1 1 0 1 1 0 1 1 0 0 1
 0 0 1 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0 0 1 1 0 1]

Score: 0.96

Selected features (0-indexed): 4, 8, 11, 15, 17, 19
```

图 5-2

5.3.2 工作原理

选择k个最好的特征，其好处在于可以处理较小维度的数据，这对减小计算复杂度来说非常有用。选择k个最佳特征的方式是基于单变量的特征选择，选择过程是先进行单变量统计测试，然后从特征向量中抽取最优秀的特征。单变量统计测试是指只涉及一个变量的分析技术。

做了这些测试后，向量空间中的每个特征将有一个评价分数。基于这些评价分数，选择最好的k个特征。我们在分类器流水线中执行这个预处理步骤。一旦抽取出k个特征，一个k维的特征向量就形成了，可以将这个特征向量用于随机森林分类器的输入训练数据。

5.4 寻找最近邻

最近邻模型是指一个通用算法类，其目的是根据训练数据集中的最近邻数量来做决策。接下来学习如何寻找最近邻。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
```

(2) 创建一些示例的二维数据：

```
# 输入数据
X = np.array([[1, 1], [1, 3], [2, 2], [2.5, 5], [3, 1],
        [4, 2], [2, 3.5], [3, 3], [3.5, 4]])
```

(3) 我们的目标是对于任意给定点找到其3个最近邻。定义该参数：

```
# 寻找最近邻的数量
num_neighbors = 3
```

(4) 定义一个随机数据点，这个数据点不包括在输入数据中：

```
# 输入数据点
input_point = [2.6, 1.7]
```

(5) 看看数据的分布，并将其画出：

```
# 画出数据点
plt.figure()
plt.scatter(X[:,0], X[:,1], marker='o', s=25, color='k')
```

(6) 为了寻找最近邻，用适合的参数定义一个`NearestNeighbors`对象，并用输入数据训练该对象：

```
# 建立最近邻模型
knn = NearestNeighbors(n_neighbors=num_neighbors, algorithm='ball_tree').fit(X)
```

(7) 计算输入点与输入数据中所有点的距离：

```
distances, indices = knn.kneighbors(input_point)
```

(8) 打印出*k*个最近邻：

```
# 打印k个最近邻点
print "\nk nearest neighbors"
for rank, index in enumerate(indices[0][:num_neighbors]):
    print str(rank+1) + " -->", X[index]
```

`indices`数组是一个已排序的数组，因此仅需要解析它并打印出数据点。

(9) 画出输入数据点，并突出显示*k*个最近邻：

```
# 画出最近邻点
plt.figure()
plt.scatter(X[:,0], X[:,1], marker='o', s=25, color='k')
plt.scatter(X[indices][0][:][:,0], X[indices][0][:][:,1],
        marker='o', s=150, color='k', facecolors='none')
plt.scatter(input_point[0], input_point[1],
        marker='x', s=150, color='k', facecolors='none')

plt.show()
```

(10) 执行代码，可以在命令行工具中看到如图5-3所示的输出。

```
k nearest neighbors
1 --> [ 2.  2.]
2 --> [ 3.  1.]
3 --> [ 3.  3.]
```

图 5-3

(11) 输入数据点的绘制情况如图5-4所示。

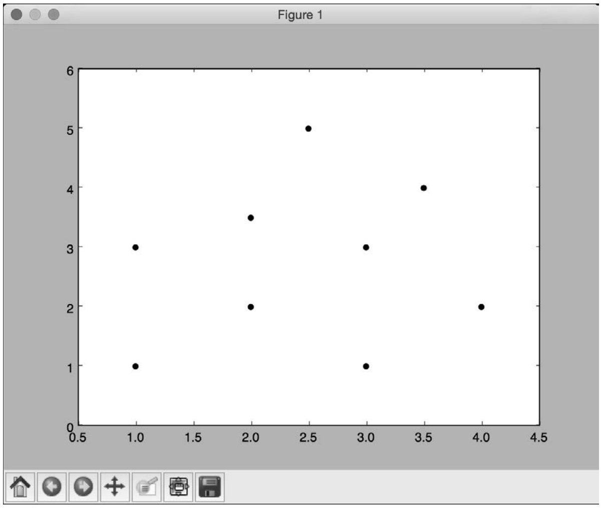


图 5-4

(12) 如图5-5所示的输出图像展示了测试数据点和3个最近邻的位置。

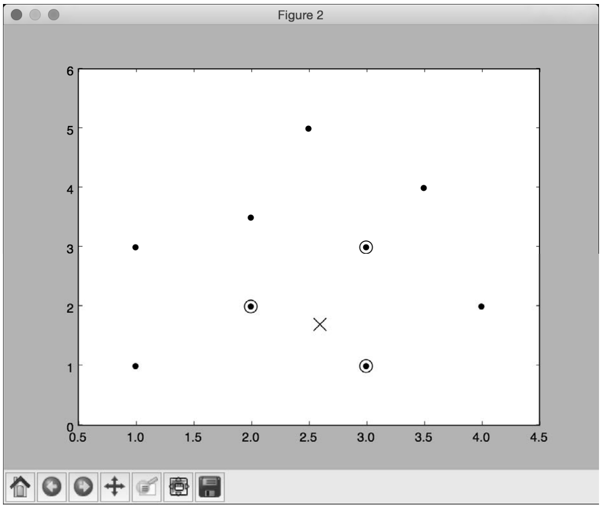


图 5-5

5.5 构建一个 KNN 分类器

KNN（k-nearest neighbors）是用*k*个最近邻的训练数据集来寻找未知对象分类的一种算法。如果希望找到未知数据点属于哪个类，可以找到KNN并做一个多数表决。接下来学习如何构建这样的分类器。

5.5.1 详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn import neighbors, datasets

from utilities import load_data
```

(2) 我们将用到data_nn_classifier.txt文件作为输入数据。加载该输入数据：

```
# 加载输入数据
input_file = 'data_nn_classifier.txt'
data = load_data(input_file)
X, y = data[:,:-1], data[:,-1].astype(np.int)
```

前两列包含输入数据，最后一列包含标签，因此分别将其用`X`和`y`两个变量表示。

(3) 将输入数据可视化：

```
# 画出输入数据
plt.figure()
plt.title('Input datapoints')
markers = '^sov<>hp'
mapper = np.array([markers[i] for i in y])
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')
```

迭代所有的数据点，并用合适的标记区分不同的类。

(4) 为了构建分类器，需要指定我们考虑的最近邻的个数。定义该参数如下：

```
# 设置最近邻的个数
num_neighbors = 10
```

(5) 为了将边界可视化，需要定义一个网格，用这个网格评价该分类器。定义网格步长如下：

```
# 定义网格步长
h = 0.01
```

(6) 接下来创建KNN分类器并进行训练：

```
# 创建KNN分类器模型并进行训练
classifier = neighbors.KNeighborsClassifier(num_neighbors, weights='distance')
classifier.fit(X, y)
```

(7) 生成一个网格来画出边界。对网格做如下定义：

```
# 建立网格来画出边界
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
x_grid, y_grid = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))
```

(8) 评价分类器对所有点的输出：

```
# 计算网格中所有点的输出
predicted_values = classifier.predict(np.c_[x_grid.ravel(), y_grid.ravel()])
```

(9) 将其画出：


```
# 在图中画出计算结果
predicted_values = predicted_values.reshape(x_grid.shape)
plt.figure()
plt.pcolormesh(x_grid, y_grid, predicted_values, cmap=cm.Pastel1)
```

(10) 我们画出了彩色网格，现在要将训练数据点覆盖在其上，查看这些点与边界的关系在哪里：

```
# 在图中画出训练数据点
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')

plt.xlim(x_grid.min(), x_grid.max())
plt.ylim(y_grid.min(), y_grid.max())
plt.title('k nearest neighbors classifier boundaries')
```

(11) 接下来测试数据点，查看分类器能否准确分类。定义并画出它：

```
# 测试输入数据点
test_datapoint = [4.5, 3.6]
plt.figure()
plt.title('Test datapoint')
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')

plt.scatter(test_datapoint[0], test_datapoint[1], marker='x',
        linewidth=3, s=200, facecolors='black')
```

(12) 用以下模型提取KNN：

```
# 提取KNN分类结果
dist, indices = classifier.kneighbors(test_datapoint)
```

(13) 画出KNN并突出显示：

```
# 画出KNN分类结果
plt.figure()
plt.title('k nearest neighbors')

for i in indices:
    plt.scatter(X[i, 0], X[i, 1], marker='o',
            linewidth=3, s=100, facecolors='black')

plt.scatter(test_datapoint[0], test_datapoint[1], marker='x',
        linewidth=3, s=200, facecolors='black')

for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')

plt.show()
```

(14) 在命令行工具中打印分类器输出结果：

```
print "Predicted output:", classifier.predict(test_datapoint)[0]
```

(15) 执行代码，第一幅输出图像展示了输入数据的分布，如图5-6所示。

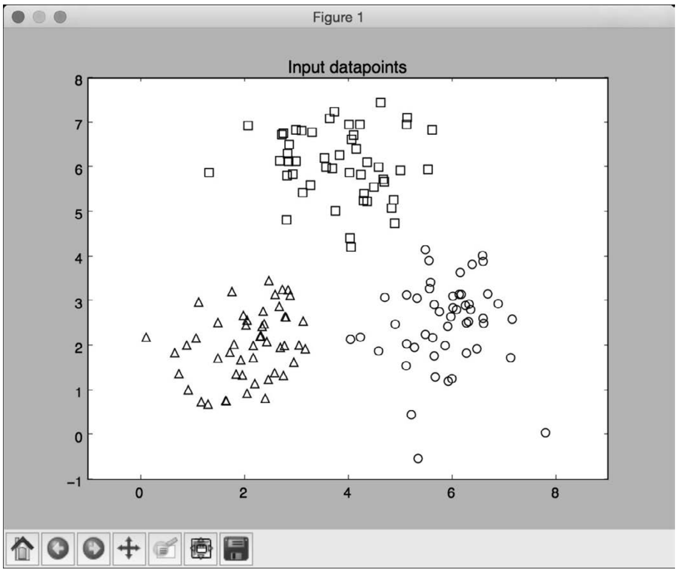


图　5-6

(16) 第二幅输出图像展示了用KNN分类器获得的边界，如图5-7所示。

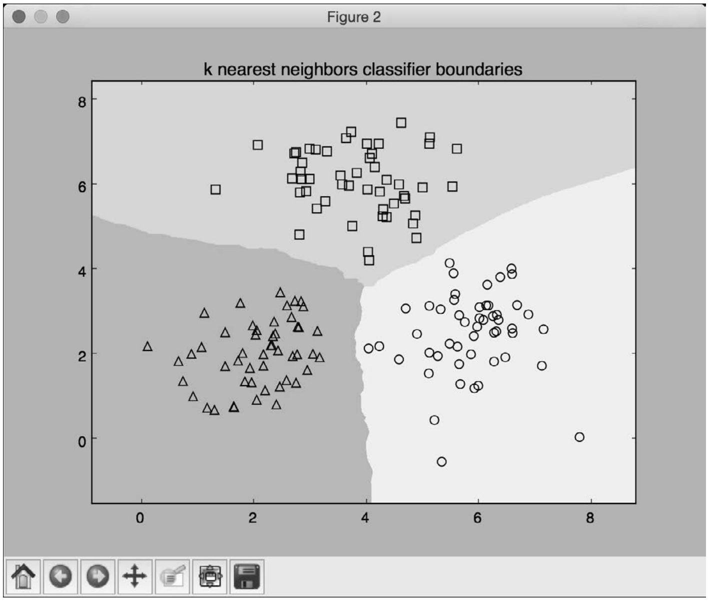


图 5-7

(17) 第三幅输出图像展示了测试数据点的位置，如图5-8所示。

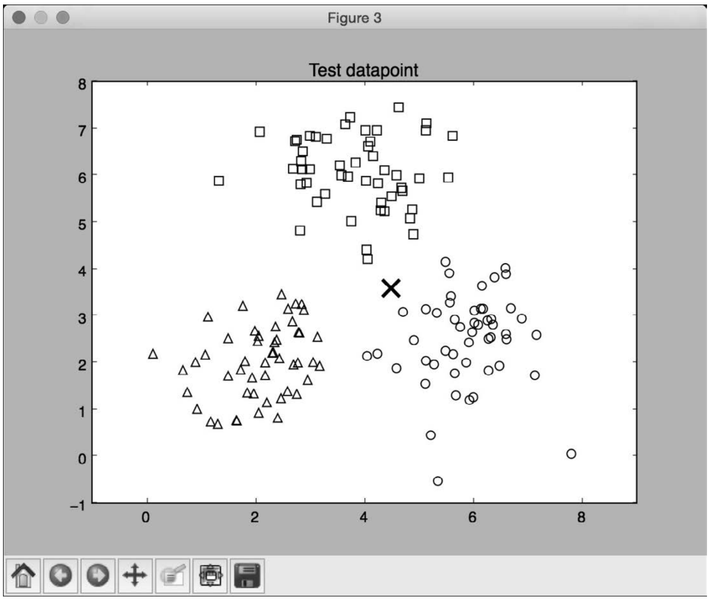


图 5-8

(18) 第四幅输出图像展示了10个最近邻的位置，如图5-9所示。

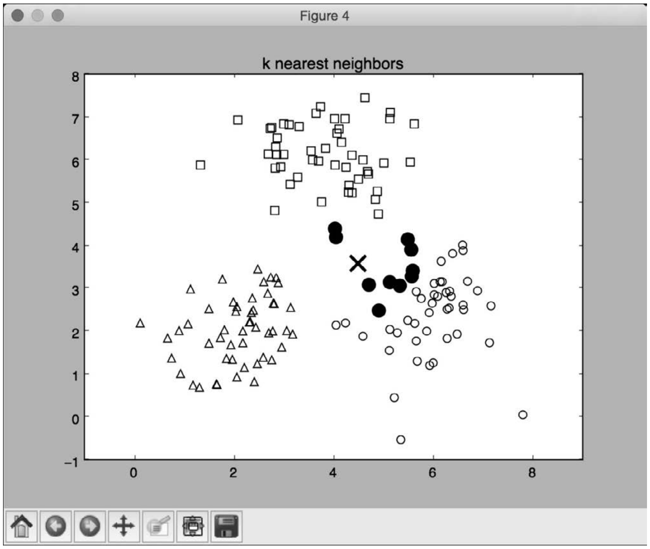


图　5-9

5.5.2　工作原理

KNN分类器存储了所有可用的数据点，并根据相似度指标来对新的数据点进行分类。这个相似度指标通常以距离函数的形式度量。该算法是一个非参数化技术，也就是说它在进行计算前并不需要找出任何隐含的参数。我们只需要选择k的值。

一旦找出KNN，就会做一个多数表决。一个新数据点通过KNN的多数表决来进行分类。这个数据点会被分到和KNN最常见的类中。如果将k的值设置为1，那么这就变成了一个最近邻分类器，在该分类器中，将数据点分类到训练数据集中其最近邻所属的哪一类。更多详细内容可查看http://www.fon.hum.uva.nl/praat/manual/kNN_classifiers_1__What_is_a_kNN_classifier_.html。

5.6　构建一个 KNN 回归器

我们已经知道如何用KNN算法构建分类器了，下面用KNN算法构建回归器。接下来学习如何构建KNN回归器。

5.6.1　详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors
```

(2) 生成一些服从正态分布的样本数据：

```
# 生成样本数据
amplitude = 10
num_points = 100
X = amplitude * np.random.rand(num_points, 1) - 0.5 * amplitude
```

(3) 下面在数据中加入一些噪声来引入一些随机性。加入噪声的目的在于测试算法是否能忽略噪声，并仍然很健壮地运行函数：

```
# 计算目标并添加噪声
y = np.sinc(X).ravel()
y += 0.2 * (0.5 - np.random.rand(y.size))
```

(4) 将数据可视化：

```
# 画出输入数据图形
plt.figure()
plt.scatter(X, y, s=40, c='k', facecolors='none')
plt.title('Input data')
```


(5) 我们在上面生成了一些数据，并针对这些点做了连续函数的评价。接下来定义更密集的网格点：

```
# 用输入数据10倍的密度创建一维网格
x_values = np.linspace(-0.5*amplitude, 0.5*amplitude, 10*num_points)[:, np.newaxis]
```

定义这个更密集的网格点，因为我们希望针对这些点评价回归器，并查看它和函数的相似程度。

(6) 定义最近邻的个数：

```
# 定义最近邻的个数
n_neighbors = 8
```

(7) 用之前定义的参数初始化并训练KNN回归器：

```
# 定义并训练回归器
knn_regressor = neighbors.KNeighborsRegressor(n_neighbors, weights='distance')
y_values = knn_regressor.fit(X, y).predict(x_values)
```

(8) 将输入数据和输出数据交叠在一起，以查看回归器的性能表现：

```
plt.figure()
plt.scatter(X, y, s=40, c='k', facecolors='none', label='input data')
plt.plot(x_values, y_values, c='k', linestyle='--', label='predicted values')
plt.xlim(X.min() - 1, X.max() + 1)
plt.ylim(y.min() - 0.2, y.max() + 0.2)
plt.aopxis('tight')
plt.legend()
plt.title('K Nearest Neighbors Regressor')

plt.show()
```

(9) 执行代码，第一幅输出图像展示了输入数据的分布，如图5-10所示。

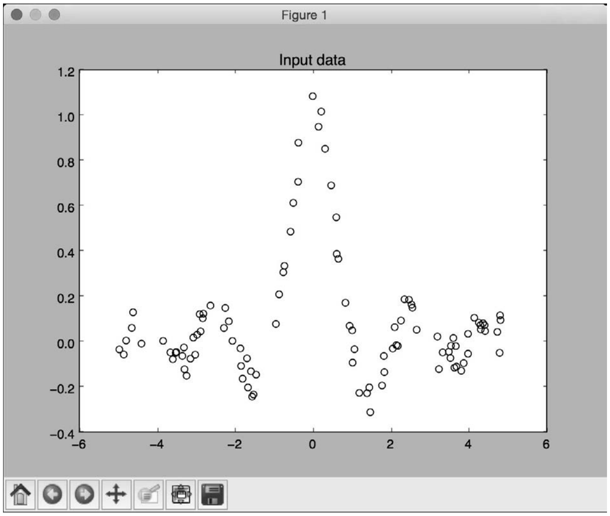


图　5-10

(10) 第二幅图像展示了用KNN回归器预测到的值，如图5-11所示。

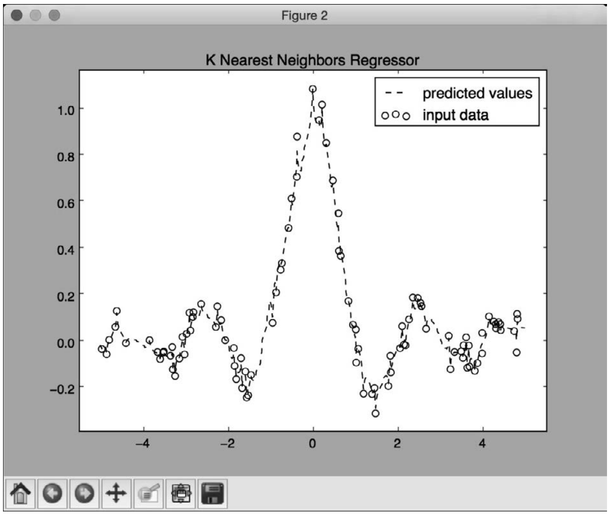


图　5-11

5.6.2　工作原理

回归器的目标是预测连续值的输出。这个例子中并没有固定数量的输出类别，仅有一组实际输出值，我们希望回归器可以预测未知数据点的输出值。这个例子中用到一个`sinc`函数来演示KNN回归器，该函数也被称为**基本正弦函数**。`sinc`函数的定义如下：

$$\mathrm{sinc}(x) = \sin(x)/x \quad (x \mathrel{!}= 0)$$
$$= 1 \quad (x = 0)$$

当x为0时，sin(x)/x变成0/0不定式，因此需要计算该函数在x无限趋近于0时函数的极限。我们用到了一组值做训练，并且定义了一个更密集的网格点来进行测试。正如在之前的图中看到的，输出曲线接近于训练输出。

5.7　计算欧氏距离分数

现在已经有了充足的机器学习流水线和最近邻分类器的背景知识，下面可以开始推荐引擎的探讨了。为了构建一个推荐引擎，需要定义相似度指标，以便找到与数据库中特定用户相似的用户。欧氏距离分数就是一个这样的指标，可以计算两个数据点之间的欧几里得距离。下面将重点讨论电影推荐引擎。接下来学习如何计算两个用户间的欧几里得分数。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import json
import numpy as np
```

(2) 定义一个用于计算两个用户之间的欧几里得分数的函数。第一步先判断用户是否在数据库中出现：

```
# 计算user1和user2的欧氏距离分数
def euclidean_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError('User ' + user1 + ' not present in the dataset')

    if user2 not in dataset:
        raise TypeError('User ' + user2 + ' not present in the dataset')
```

(3) 为了计算分数，需要提取两个用户均评过分的电影：

```
    # 提取两个用户均评过分的电影
    rated_by_both = {}

    for item in dataset[user1]:
        if item in dataset[user2]:
            rated_by_both[item] = 1
```

(4) 如果没有两个用户共同评过分的电影，则说明这两个用户之间没有相似度（至少根据数据库中的评分信息无法计算出来）：

```
    # 如果两个用户都没评分过，得分为0
```

```
    if len(rated_by_both) == 0:
        return 0
```

(5) 对于每个共同评分，只计算平方和的平方根，并将该值归一化，使得评分取值在0到1之间：

```
    squared_differences = []

    for item in dataset[user1]:
        if item in dataset[user2]:
            squared_differences.append(np.square(dataset[user1][item] -
dataset[user2][item]))

    return 1 / (1 + np.sqrt(np.sum(squared_differences)))
```

如果评分相似，那么平方和的差别就会很小，因此评分就会变得很高，这也是我们希望指标达到的效果。

(6) 加载数据文件中的movie_ratings.json文件：

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())
```

(7) 假定两个随机用户，计算其欧氏距离分数：

```
    user1 = 'John Carson'
    user2 = 'Michelle Peterson'

    print "\nEuclidean score:"
    print euclidean_score(data, user1, user2)
```

(8) 运行该代码，可以看到欧氏距离分数显示在命令行工具中。

5.8　计算皮尔逊相关系数

欧氏距离分数是一个非常好的指标，但它也有一些缺点。因此，皮尔逊相关系数常用于推荐引擎。接下来学习如何计算皮尔逊相关系数。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import json
import numpy as np
```

(2) 接下来定义一个用于计算两个用户之间的皮尔逊相关度系数的函数。第一步先判断用户

是否在数据库中出现：

```
# 计算user1和user2的皮尔逊相关系数
def pearson_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError('User ' + user1 + ' not present in the dataset')

    if user2 not in dataset:
        raise TypeError('User ' + user2 + ' not present in the dataset')
```

(3) 提取两个用户均评过分的电影：

```
    # 提取两个用户均评过分的电影
    rated_by_both = {}

    for item in dataset[user1]:
        if item in dataset[user2]:
            rated_by_both[item] = 1

    num_ratings = len(rated_by_both)
```


(4) 如果没有两个用户共同评过分的电影，则说明这两个用户之间没有相似度，此时返回0：

```
    # 如果两个用户都没有评分，得分为0
    if num_ratings == 0:
        return 0
```

(5) 计算相同评分电影的平方值之和：

```
    # 计算相同评分电影的平方值之和
    user1_sum = np.sum([dataset[user1][item] for item in rated_by_both])
    user2_sum = np.sum([dataset[user2][item] for item in rated_by_both])
```

(6) 计算所有相同评分电影的评分的平方和：

```
    # 计算所有相同评分电影的评分的平方和
    user1_squared_sum = np.sum([np.square(dataset[user1][item]) for item in
rated_by_both])
    user2_squared_sum = np.sum([np.square(dataset[user2][item]) for item in
rated_by_both])
```

(7) 计算数据集的乘积之和：

```
    # 计算数据集的乘积之和
    product_sum = np.sum([dataset[user1][item] * dataset[user2][item] for item in
rated_by_both])
```

(8) 计算皮尔逊相关系数需要的各种元素：

```
    # 计算皮尔逊相关度
    Sxy = product_sum - (user1_sum * user2_sum / num_ratings)
    Sxx = user1_squared_sum - np.square(user1_sum) / num_ratings
    Syy = user2_squared_sum - np.square(user2_sum) / num_ratings
```

(9) 考虑分母为0的情况：

```
    if Sxx * Syy == 0:
        return 0
```

(10) 如果一切正常，返回皮尔逊相关系数：

```
    return Sxy / np.sqrt(Sxx * Syy)
```

(11) 定义main函数并计算两个用户之间的皮尔逊相关系数：

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())

    user1 = 'John Carson'
    user2 = 'Michelle Peterson'

    print "\nPearson score:"
    print pearson_score(data, user1, user2)
```

(12) 运行该代码，可以看到皮尔逊相关系数显示在命令行工具中。

5.9　寻找数据集中的相似用户

构建推荐引擎中一个非常重要的任务是寻找相似的用户，也就是说，为某位用户生成的推荐信息可以同时推荐给与其相似的用户。接下来学习如何寻找相似用户。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import json
import numpy as np

from pearson_score import pearson_score
```

(2) 定义一个函数，用于寻找与输入用户相似的用户。该函数有3个输入参数：数据库、输入用户和寻找的相似用户个数。第一步是查看该用户是否包含在数据库中。如果用户已经存在，则需要计算该用户与数据库中其他所有用户的皮尔逊相关系数：

```
# 寻找特定数量的与输入用户相似的用户
def find_similar_users(dataset, user, num_users):
    if user not in dataset:
        raise TypeError('User ' + user + ' not present in the dataset')
```

```
    # 计算所有用户的皮尔逊相关度
    scores = np.array([[x, pearson_score(dataset, user, x)] for x in dataset if user !=
x])
```

(3) 将这些得分按照降序排列：

```
    # 评分按照第二列排序
    scores_sorted = np.argsort(scores[:, 1])

    # 评分按照降序排列
    scored_sorted_dec = scores_sorted[::-1]
```

(4) 提取出*k*个最高分并返回：

```
    # 提取出k个最高分
    top_k = scored_sorted_dec[0:num_users]

    return scores[top_k]
```


(5) 定义`main`函数，加载输入数据库：

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())
```

(6) 我们希望查找3个与`John Carson`相似的用户。用以下步骤实现：

```
    user = 'John Carson'
    print "\nUsers similar to " + user + ":\n"
    similar_users = find_similar_users(data, user, 3)
    print "User\t\t\tSimilarity score\n"
    for item in similar_users:
        print item[0], '\t\t', round(float(item[1]), 2)
```

(7) 运行该代码，可以看到命令行工具中显示了3个用户，如图5-12所示。

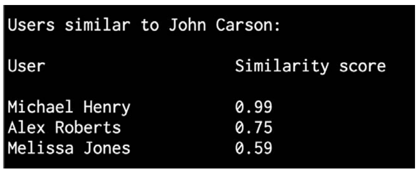


图 5-12

5.10 生成电影推荐

至此，我们已经创建了一个推荐引擎的各个不同组成部分，接下来生成一个实际的电影推荐系统。本节将用到前面各节中构建的函数来构建电影推荐引擎，接下来看看具体如何实现。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import json
import numpy as np

from euclidean_score import euclidean_score
from pearson_score import pearson_score
from find_similar_users import find_similar_users
```

(2) 定义一个为给定用户生成电影推荐的函数。首先检查该用户是否存在于数据库中：

```
# 为给定用户生成电影推荐
def generate_recommendations(dataset, user):
    if user not in dataset:
        raise TypeError('User ' + user + ' not present in the dataset')
```

(3) 计算该用户与数据库中其他用户的皮尔逊相关系数：

```
    total_scores = {}
    similarity_sums = {}

    for u in [x for x in dataset if x != user]:
        similarity_score = pearson_score(dataset, user, u)

        if similarity_score <= 0:
            continue
```

(4) 找到还未被该用户评分的电影：

```
        for item in [x for x in dataset[u] if x not in dataset[user] or dataset[user][x]
== 0]:
            total_scores.update({item: dataset[u][item] * similarity_score})
            similarity_sums.update({item: similarity_score})
```

(5) 如果该用户看过数据库中所有的电影，那就不能为用户推荐电影。对该条件做如下处理：

```
    if len(total_scores) == 0:
        return ['No recommendations possible']
```

(6) 有了皮尔逊相关系数列表，下面生成一个电影评分标准化列表：

```
    # 生成一个电影评分标准化列表
    movie_ranks = np.array([[total/similarity_sums[item], item]
            for item, total in total_scores.items()])
```

(7) 对皮尔逊相关系数进行降序排列：

```
    # 根据第一列对皮尔逊相关系数进行降序排列
    movie_ranks = movie_ranks[np.argsort(movie_ranks[:, 0])[::-1]]
```

(8) 最后提取出推荐的电影：

```
    # 提取出推荐的电影
    recommendations = [movie for _, movie in movie_ranks]

    return recommendations
```

(9) 定义`main`函数，加载数据集：

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())
```

(10) 为`Michael Henry`生成推荐：

```
    user = 'Michael Henry'
    print "\nRecommendations for " + user + ":"
    movies = generate_recommendations(data, user)
    for i, movie in enumerate(movies):
        print str(i+1) + '. ' + movie
```

(11) 用户`John Carson`看过所有电影，因此在为他推荐电影时，应该显示0推荐。来看看程序运行结果是否正确：

```
    user = 'John Carson'
    print "\nRecommendations for " + user + ":"
    movies = generate_recommendations(data, user)
    for i, movie in enumerate(movies):
        print str(i+1) + '. ' + movie
```

(12) 运行该代码，可以在命令行工具中看到如图5-13所示的结果。

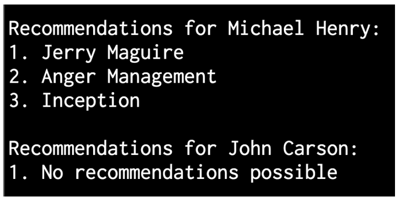


图 5-13

第 6 章

分析文本数据

在这一章，我们将介绍以下主题：

- 用标记解析的方法预处理数据
- 提取文本数据的词干
- 用词形还原的方法还原文本的基本形式
- 用分块的方法划分文本
- 创建词袋模型（bag-of-words model）
- 创建文本分类器
- 识别性别
- 分析句子的情感
- 用主题建模识别文本的模式

6.1 简介

文本分析和NLP（Natural Language Processing，自然语言处理）是现代人工智能系统不可分割的一部分。计算机擅长于用有限的多样性来理解结构死板的数据。然而，当我们用计算机处理非结构化的自由文本时，就会变得很困难。开发NLP应用程序是一种挑战，因为计算机很难理解隐含的概念，而且语言交流方式也有很多细微的差异。这些差异的形式可以是方言、语境、俚语等。

为了解决这个问题，基于机器学习的NLP应运而生。这些算法检测文本数据的模式，以便可以从中得到了解。人工智能公司大量地使用了NLP和文本分析来推送相关结果。NLP最常用的领域包括搜索引擎、情感分析、主题建模、词性标注、实体识别等。NLP的目标是开发出一组算法，以便可以用简单的英文和计算机交流。如果这一目标实现，将不再需要程序设计语言来命令计算机执行指令。这一章将主要介绍文本分析，以及如何从文本数据中提取有意义的信息。我们将大量用到Python中的NLTK（Natural Language Toolkit）包。在进行接下来的学习之前，先确保你已经安装了NLTK，安装步骤可以参考http://www.nltk.org/install.html。你还需要安装NLTK数据，这

些数据中包含很多语料和训练模型，这也是文本分析不可分割的部分，安装步骤可以参考 http://www.nltk.org/data.html。

6.2 用标记解析的方法预处理数据

标记解析是将文本分割成一组有意义的片段的过程。这些片段被称作**标记**，例如可以将一段文字分割成单词或者句子。根据手头的任务需要，可以自定义将输入的文本分割成有意义的标记。接下来介绍如何实现这样的标记解析。

详细步骤

(1) 创建一个Python文件，在文件中加入以下内容。这里定义了一些用于分析的示例文本：

```
text = "Are you curious about tokenization? Let's see how it works!  We need to analyze
a couple of sentences with punctuations to see it in action."
```

(2) 接下来做句子解析。NLTK提供了一个句子解析器，首先加载该模块：

```
# 对句子进行解析
from nltk.tokenize import sent_tokenize
```


(3) 对输入文本运行句子解析器，提取出标记：

```
sent_tokenize_list = sent_tokenize(text)
```

(4) 打印出句子解析结果列表：

```
print "\nSentence tokenizer:"
print sent_tokenize_list
```

(5) 单词解析在NLP中是非常常用的。NLTK附带了几个不同的单词解析器。先从最基本的单词解析器开始：

```
# 建立一个新的单词解析器
from nltk.tokenize import word_tokenize

print "\nWord tokenizer:"
print word_tokenize(text)
```

(6) NLTK中另外一个可以使用的单词解析器叫`PunktWord`，它以标点符号分割文本，如果是单词中的标点符号，则保留不做分割：

```
# 创建一个带标点的单词解析器
from nltk.tokenize import PunktWordTokenizer

punkt_word_tokenizer = PunktWordTokenizer()
print "\nPunkt word tokenizer:"
print punkt_word_tokenizer.tokenize(text)
```

(7) 如果需要将标点符号保留到不同的句子标记中，可以用WordPunct标记解析器：

```
# 创建一个新的WordPunct标记解析器
from nltk.tokenize import WordPunctTokenizer

word_punct_tokenizer = WordPunctTokenizer()
print "\nWord punct tokenizer:"
print word_punct_tokenizer.tokenize(text)
```

(8) 全部代码已经在tokenizer.py文件中给出。运行该代码，可以在命令行工具中看到如图6-1所示的输出结果。

```
Sentence tokenizer:
['Are you curious about tokenization?', "Let's see how it works!", 'We need to analyze a couple of senten
ces with punctuations to see it in action.']

Word tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'s", 'see', 'how', 'it', 'works', '!', 'W
e', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it',
 'in', 'action', '.']

Punkt word tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'s", 'see', 'how', 'it', 'works', '!', 'W
e', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it',
 'in', 'action.']

Word punct tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'", 's', 'see', 'how', 'it', 'works', '!'
, 'We', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', '
it', 'in', 'action', '.']
```

图　6-1

6.3　提取文本数据的词干

处理文本文档时，可能会碰到单词的不同形式。以单词“play”为例，这个单词可能以各种形式出现，例如“play”“plays”“player”“playing”等，这些是具有同样含义的单词家族。在文本分析中，提取这些单词的原形非常有用，它有助于我们提取一些统计信息来分析整个文本。词干提取的目标是将不同词形的单词都变成其原形。词干提取使用启发式处理方法截取单词的尾部，以提取单词的原形。接下来介绍如何在Python中完成词干提取。

6.3.1　详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem.snowball import SnowballStemmer
```

(2) 定义一些单词来进行词干提取：

```
words = ['table', 'probably', 'wolves', 'playing', 'is','dog', 'the', 'beaches',
'grounded', 'dreamt', 'envision']
```

(3) 定义一个稍后会用到的词干提取器列表：

```
# 对比不同的词干提取算法
stemmers = ['PORTER', 'LANCASTER', 'SNOWBALL']
```

(4) 初始化3个词干提取器对象：

```
stemmer_porter = PorterStemmer()
stemmer_lancaster = LancasterStemmer()
stemmer_snowball = SnowballStemmer('english')
```

(5) 为了以整齐的表格形式将输出数据打印出来，需要设定其正确的格式：

```
formatted_row = '{:>16}' * (len(stemmers) + 1)
print '\n', formatted_row.format('WORD', *stemmers), '\n'
```

(6) 迭代列表中的单词，并用3个词干提取器进行词干提取：

```
for word in words:
    stemmed_words = [stemmer_porter.stem(word),
            stemmer_lancaster.stem(word), stemmer_snowball.stem(word)]
    print formatted_row.format(word, *stemmed_words)
```


(7) 全部代码已经在stemmer.py文件中给出。运行该代码，可以在命令行工具中看到如图6-2所示的输出结果。从图6-2中可以看出，Lancaster词干提取器的输出结果与其他词干提取器的输出结果不同。

```
            WORD          PORTER       LANCASTER        SNOWBALL

           table            tabl            tabl            tabl
        probably         probabl            prob         probabl
          wolves            wolv            wolv            wolv
         playing            play            play            play
              is              is              is              is
             dog             dog             dog             dog
             the             the             the             the
         beaches           beach           beach           beach
        grounded          ground          ground          ground
          dreamt          dreamt          dreamt          dreamt
        envision           envis           envid           envis
```

图　6-2

6.3.2　工作原理

以上3种词干提取算法的本质目标都是提取出词干，消除词形的影响。它们的不同之处在于操作的严格程度不同。观察输出结果可以看到，Lancaster词干提取器比其他两个词干提取器更严

格。就严格程度来说，Porter词干提取器是最宽松的，而Lancaster词干提取器是最严格的。从Lancaster词干提取器得到的词干往往比较模糊，难以理解。Lancaster词干提取算法的速度很快，但是它会减少单词的很大部分，因此通常会选择Snowball词干提取器。

6.4　用词形还原的方法还原文本的基本形式

词形还原的目标也是将单词转换为其原形，但它是一个更结构化的方法。在前一节中，可以看到用词根还原得到的单词原形并不是有意义的，例如单词"wolves"被还原成"wolv"，还原出的单词根本不是一个真实的单词。词形还原通过对单词进行词汇和语法分析来实现，因此可以圆满解决这一问题。词形还原变形词的结尾，例如"ing"或"ed"，然后返回单词的原形形式，这个原形也就是词根（lemma）。如果对单词"wolves"做词根还原，可以得到"wolf"的输出。输出结果取决于标记是一个动词还是一个名词。下面看看如何做词形还原。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
from nltk.stem import WordNetLemmatizer
```

(2) 定义一组单词来进行词形还原：

```
words = ['table', 'probably', 'wolves', 'playing', 'is',
        'dog', 'the', 'beaches', 'grounded', 'dreamt', 'envision']
```

(3) 比较两个不同的词形还原器，`NOUN`词形还原器和`VERB`词形还原器：

```
#  对比不同的词形还原器
lemmatizers = ['NOUN LEMMATIZER', 'VERB LEMMATIZER']
```

(4) 基于`WordNet`词形还原器创建一个对象：

```
lemmatizer_wordnet = WordNetLemmatizer()
```

(5) 为了以整齐的表格形式将输出数据打印出来，需要设定其正确的格式：

```
formatted_row = '{:>24}' * (len(lemmatizers) + 1)
print '\n', formatted_row.format('WORD', *lemmatizers), '\n'
```

(6) 迭代列表中的单词，并用词形还原器进行词形还原：

```
for word in words:
    lemmatized_words = [lemmatizer_wordnet.lemmatize(word, pos='n'),
           lemmatizer_wordnet.lemmatize(word, pos='v')]
    print formatted_row.format(word, *lemmatized_words)
```

(7) 全部代码已经在lemmatizer.py文件中给出。运行该代码，可以看到如图6-3所示的输出结果。

观察图中对于单词“is”的词形还原，NOUN 词形还原器和VERB词形还原器的还原结果有何不同。

WORD	NOUN LEMMATIZER	VERB LEMMATIZER
table	table	table
probably	probably	probably
wolves	wolf	wolves
playing	playing	play
is	is	be
dog	dog	dog
the	the	the
beaches	beach	beach
grounded	grounded	ground
dreamt	dreamt	dream
envision	envision	envision

图 6-3

6.5 用分块的方法划分文本

分块是指基于任意随机条件将输入文本分割成块。与标记解析不同的是，分块没有条件约束，分块的结果不需要有实际意义。分块在文本分析中经常使用。当处理非常大的文本文档时，就需要将文本进行分块，以便进行下一步分析。在这一节中，将输入文本分成若干块，每块都包含固定数目的单词。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import numpy as np
from nltk.corpus import brown
```

(2) 定义一个将文本分割成块的函数。第一步是将文本按照空格划分：

```
# 将文本分割成块
def splitter(data, num_words):
    words = data.split(' ')
    output = []
```

(3) 初始化一些后面需要用到的变量：

```
    cur_count = 0
    cur_words = []
```

(4) 对这些单词进行迭代：

```
    for word in words:
        cur_words.append(word)
        cur_count += 1
```

(5) 获得的单词数量与所需的单词数量相等时，重置相应的变量：

```
        if cur_count == num_words:
            output.append(' '.join(cur_words))
            cur_words = []
            cur_count = 0
```

(6) 将块添加到输出变量列表的最后，并返回该输出变量：

```
    output.append(' '.join(cur_words) )

    return output
```

(7) 定义一个main函数，从布朗语料库（Brown corpus）加载数据。用到前10 000个单词：

```
if __name__=='__main__':
    # 从布朗语料库加载数据
    data = ' '.join(brown.words()[:10000])
```

(8) 定义每块包含的单词数目：

```
    # 每块包含的单词数目
    num_words = 1700
```

(9) 定义两个相关变量：

```
    chunks = []
    counter = 0
```

(10) 对这个文本数据调用splitter函数，并将其打印输出：

```
    text_chunks = splitter(data, num_words)

    print "Number of text chunks =", len(text_chunks)
```

(11) 全部代码已经在chunking.py文件中给出。运行该代码，可以看到生成的块的数目被打印在命令行工具中，并且块的数目应该是6。

6.6　创建词袋模型

如果需要处理包含数百万单词的文本文档，需要将其转化成某种数值表示形式，以便让机器用这些数据来学习算法。这些算法需要数值数据，以便可以对这些数据进行分析，并输出有用的信息。这里需要用到**词袋**（bag-of-words）。词袋是从所有文档的所有单词中学习词汇的模型。学习之后，词袋通过构建文档中所有单词的直方图来对每篇文档进行建模。

6.6.1　详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import numpy as np
from nltk.corpus import brown
from chunking import splitter
```

(2) 定义一个main函数，从布朗语料库加载输入数据：

```
if __name__=='__main__':
    # 布朗语料库读取数据
    data = ' '.join(brown.words()[:10000])
```

(3) 将文本数据分成5块：

```
    # 每块包含的单词数量
    num_words = 2000

    chunks = []
    counter = 0

    text_chunks = splitter(data, num_words)
```

(4) 创建一个基于这些文本块的词典：

```
    for text in text_chunks:
        chunk = {'index': counter, 'text': text}
        chunks.append(chunk)
        counter += 1
```


(5) 下一步是提取一个文档-词矩阵。文档-词矩阵记录了文档中每个单词出现的频次。下面将用scikit-learn来构建这样的矩阵，因为相比于NLTK，scikit-learn有更简洁的完成这一任务的方式。导入以下程序包：

```
    # 提取文档-词矩阵
    from sklearn.feature_extraction.text import CountVectorizer
```

(6) 定义对象，并提取文档-词矩阵:

```
    vectorizer = CountVectorizer(min_df=5, max_df=.95)
    doc_term_matrix = vectorizer.fit_transform([chunk['text'] for chunk in chunks])
```

(7) 从vectorizer对象中提取词汇，并打印出：

```
    vocab = np.array(vectorizer.get_feature_names())
    print "\nVocabulary:"
    print vocab
```

(8) 打印文档-词矩阵:

```
    print "\nDocument term matrix:"
    chunk_names = ['Chunk-0', 'Chunk-1', 'Chunk-2', 'Chunk-3', 'Chunk-4']
```

(9) 为了打印成表格形式，可以做如下格式设置：

```
    formatted_row = '{:>12}' * (len(chunk_names) + 1)
    print '\n', formatted_row.format('Word', *chunk_names), '\n'
```

(10) 迭代所有单词，打印每个单词出现在不同块中的次数：

```
for word, item in zip(vocab, doc_term_matrix.T):
    # “item”是压缩的稀疏矩阵（csr_matrix）数据结构
    output = [str(x) for x in item.data]
    print formatted_row.format(word, *output)
```

(11) 全部代码已经在bag_of_words.py文件中给出。运行该代码，可以看到命令行工具中有两部分输出，第一部分输出的是词汇，如图6-4所示。

```
Vocabulary:
[u'about' u'after' u'against' u'aid' u'all' u'also' u'an' u'and' u'are'
 u'as' u'at' u'be' u'been' u'before' u'but' u'by' u'committee' u'congress'
 u'did' u'each' u'education' u'first' u'for' u'from' u'general' u'had'
 u'has' u'have' u'he' u'health' u'his' u'house' u'in' u'increase' u'is'
 u'it' u'last' u'made' u'make' u'may' u'more' u'no' u'not' u'of' u'on'
 u'one' u'only' u'or' u'other' u'out' u'over' u'pay' u'program' u'proposed'
 u'said' u'similar' u'state' u'such' u'take' u'than' u'that' u'the' u'them'
 u'there' u'they' u'this' u'time' u'to' u'two' u'under' u'up' u'was'
 u'were' u'what' u'which' u'who' u'will' u'with' u'would' u'year' u'years']
```

图　6-4

(12) 第二部分输出的是文档-词矩阵，这个矩阵非常大，这里仅显示前几行，如图6-5所示。

```
Document term matrix:

       Word     Chunk-0     Chunk-1     Chunk-2     Chunk-3     Chunk-4

      about           1           1           1           1           3
      after           2           3           2           1           3
    against           1           2           2           1           1
        aid           1           1           1           3           5
        all           2           2           5           2           1
       also           3           3           3           4           3
         an           5           7           5           7          10
        and          34          27          36          36          41
        are           5           3           6           3           2
         as          13           4          14          18           4
         at           5           7           9           3           6
         be          20          14           7          10          18
       been           7           1           6          15           5
     before           2           2           1           1           2
        but           3           3           2           9           5
         by           8          22          15          14          12
  committee           2          10           3           1           7
```

图　6-5

6.6.2　工作原理

以下面的句子为例：

- 句子1：The brown dog is running.
- 句子2：The black dog is in the black room.
- 句子3： Running in the room is forbidden.

以上3句话中有以下9个唯一的单词：

- the
- brown
- dog
- is
- running
- black
- in
- room
- forbidden

我们利用这些单词在每个句子中出现的频次将每个句子转换成直方图。每一个特征向量均是9维的，因为唯一单词个数为9：

- 句子1：[1, 1, 1, 1, 1, 0, 0, 0, 0]
- 句子2：[2, 0, 1, 1, 0, 2, 1, 1, 0]
- 句子3：[0, 0, 0, 1, 1, 0, 1, 1, 1]

提取出这样的特征向量后，就可以用机器学习的算法对这些向量进行分析了。

6.7 创建文本分类器

文本分类的目的是将文本文档分为不同的类，这是NLP中非常重要的分析手段。这里将使用一种技术，它基于一种叫作tf-idf的统计数据，它表示**词频–逆文档频率**（term frequency—inverse document frequency）。这个统计工具有助于理解一个单词在一组文档中对某一个文档的重要性。它可以作为特征向量来做文档分类，你可以在http://www.tfidf.com中找到更多详细介绍。

6.7.1 详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
from sklearn.datasets import fetch_20newsgroups
```

(2) 选择一个类型列表，并且用词典映射的方式定义。这些类型是加载的新闻组数据集的一部分：

```
category_map = {'misc.forsale': 'Sales', 'rec.motorcycles': 'Motorcycles',
        'rec.sport.baseball': 'Baseball', 'sci.crypt': 'Cryptography',
        'sci.space': 'Space'}
```

(3) 基于刚刚定义的类型加载训练数据：

```
training_data = fetch_20newsgroups(subset='train', categories=category_map.keys(),
shuffle=True, random_state=7)
```

(4) 导入特征提取器：

```
# 特征提取
from sklearn.feature_extraction.text import CountVectorizer
```

(5) 用训练数据提取特征：

```
vectorizer = CountVectorizer()
X_train_termcounts = vectorizer.fit_transform(training_data.data)
print "\nDimensions of training data:", X_train_termcounts.shape
```

(6) 接下来训练分类器。这里将用到多项式朴素贝叶斯（Multinomial Naive Bayes）分类器：

```
# 训练分类器
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfTransformer
```

(7) 定义一些随机输入的句子：

```
input_data = [
    "The curveballs of right handed pitchers tend to curve to the left",
    "Caesar cipher is an ancient form of encryption",
    "This two-wheeler is really good on slippery roads"
]
```

(8) 定义tf-idf变换器对象，并训练：

```
# tf-idf变换器
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_ termcounts)
```

(9) 得到特征向量后，用该数据训练多项式朴素贝叶斯分类器：

```
# 多项式朴素贝叶斯分类器
classifier = MultinomialNB().fit(X_train_tfidf, training_data. target)
```

(10) 用词频统计转换输入数据：

```
X_input_termcounts = vectorizer.transform(input_data)
```

(11) 用tf-idf变换器变换输入数据：

```
X_input_tfidf = tfidf_transformer.transform(X_input_termcounts)
```

(12) 用训练过的分类器预测这些输入句子的输出类型：

```
# 预测输出类型
predicted_categories = classifier.predict(X_input_tfidf)
```

(13) 打印输出：

```
# 打印输出
for sentence, category in zip(input_data, predicted_categories):
    print '\nInput:', sentence, '\nPredicted category:', \
            category_map[training_data.target_names[category]]
```

(14) 全部代码已经在tfidf.py文件中给出。运行该代码，可以在命令行工具中看到如图6-6所示的输出结果。

```
Dimensions of training data: (2968, 40605)

Input: The curveballs of right handed pitchers tend to curve to the left
Predicted category: Baseball

Input: Caesar cipher is an ancient form of encryption
Predicted category: Cryptography

Input: This two-wheeler is really good on slippery roads
Predicted category: Motorcycles
```

图　6-6

6.7.2　工作原理

tf-idf技术常用于信息检索领域，目的是了解文档中每个单词的重要性。如果想要识别在文档中多次出现的单词，同时，像“is”和“be”这样的普通词汇并不能真正反映内容的本质，因此仅需要提取出具有实际意义的那些单词。词频越大，则表示这个词越重要，同时，如果这个词经常出现，那么这个词的词频也会增加，这两个因素互相平衡。提取出每个句子的词频，然后将其转换为特征向量，用分类器来对这些句子进行分类。

词频（The term frequency, TF）表示一个单词在给定文档中出现的频次。由于不同文档的长度不同，这些频次的直方图看起来会相差很大，因此需要将其规范化，使得这些频次可以在平等的环境下进行对比。为了实现规范化，我们用频次除以文档中所有单词的个数。**逆文档频率**（inverse document frequency，IDF）表示给定单词的重要性。当需要计算词频时，假定所有单词是同等重要的。为了抗衡哪些经常出现的单词的频率，需要用一个系数将其权重变小。我们需要计算文档的总数目除以该单词出现的文档数目的比值。逆文档频率对该比值取对数值。

例如，“is”或“the”等简单的单词在各个文档中均大量出现，但是这并不意味着要用这些词作为该篇文档的特征。同时，如果一个单词仅出现一次，那这个单词也不是十分有意义。因此，我们寻找的是那些出现了一定次数，但不太频繁以至于变成噪声的单词。tf-idf值的计算可以挑选出符合要求的单词，并且可以用于分类文档。搜索引擎经常用tf-idf工具来对搜索结果进行相关度排序。

6.8　识别性别

在NLP中，通过姓名识别性别是一个很有趣的任务。这里将用到启发式方法，即姓名的最后几个字符可以界定性别特征。例如，如果某一个名字以“la”结尾，那么它很有可能是一个女性名字，如“Angela”或者“Layla”。另外，如果一个名字以“im”结尾，那么它很有可能是一个男性名字，例如“Tim”或者“Jim”。确定需要用到几个字符来确定性别后，可以来做这个实验。接下来介绍如何识别性别。

详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import random
from nltk.corpus import names
from nltk import NaiveBayesClassifier
from nltk.classify import accuracy as nltk_accuracy
```

(2) 定义一个用于提取输入单词的特征的函数：

```
# 提取输入单词的特征
def gender_features(word, num_letters=2):
    return {'feature': word[-num_letters:].lower()}
```

(3) 定义main函数，需要一些带标记的训练数据：

```
if __name__=='__main__':
    # 提取标记名称
    labeled_names = ([(name, 'male') for name in names.words('male.txt')] +
            [(name, 'female') for name in names.words('female.txt')])
```

(4) 设置随机生成数的种子值，并混合搅乱训练数据：

```
    random.seed(7)
    random.shuffle(labeled_names)
```

(5) 定义一些输入的姓名：

```
    input_names = ['Leonardo', 'Amy', 'Sam']
```

(6) 因为不知道需要多少个末尾字符，这里将这个参数设置为1~5。每次循环执行，都会截取相应大小的末尾字符个数：

```
    # 搜索参数空间
    for i in range(1, 5):
        print '\nNumber of letters:', i
        featuresets = [(gender_features(n, i), gender) for (n, gender) in
labeled_names]
```

(7) 将数据分为训练数据集和测试数据集：

```
train_set, test_set = featuresets[500:], featuresets[:500]
```

(8) 用朴素贝叶斯分类器做分类：

```
classifier = NaiveBayesClassifier.train(train_set)
```

(9) 用参数空间的每一个值评价分类器的效果

```
# 打印分类器的准确性
print 'Accuracy ==>', str(100 * nltk_accuracy(classifier, test_set)) + str('%')

# 为新输入预测输出结果
for name in input_names:
    print name, '==>', classifier.classify(gender_features(name, i))
```

(10) 全部代码已经在gender_identification.py文件中给出。运行该代码，可以看到命令行工具中如图6-7所示的输出结果。

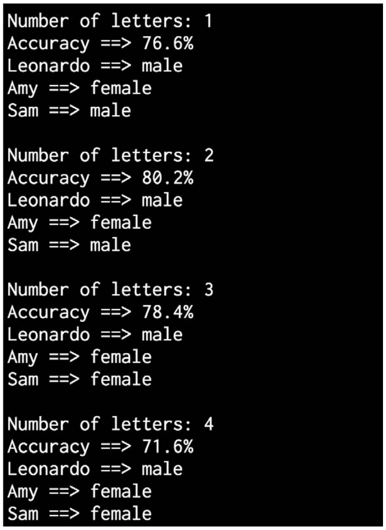


图　6-7

6.9　分析句子的情感

情感分析是NLP最受欢迎的应用之一。情感分析是指确定一段给定的文本是积极还是消极的过程。有一些场景中，我们还会将“中性”作为第三个选项。情感分析常用于发现人们对于一个特定主题的看法。情感分析用于分析很多场景中用户的情绪，如营销活动、社交媒体、电子商务客户等。

6.9.1　详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
import nltk.classify.util
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews
```

(2) 定义一个用于提取特征的函数：

```
def extract_features(word_list):
    return dict([(word, True) for word in word_list])
```

(3) 我们需要训练数据，这里将用NLTK提供的电影评论数据：

```
if __name__=='__main__':
    # 加载积极与消极评论
    positive_fileids = movie_reviews.fileids('pos')
    negative_fileids = movie_reviews.fileids('neg')
```

(4) 将这些评论数据分成积极评论和消极评论：

```
    features_positive = [(extract_features(movie_reviews.words(fileids=[f])),
            'Positive') for f in positive_fileids]
    features_negative = [(extract_features(movie_reviews.words(fileids=[f])),
            'Negative') for f in negative_fileids]
```

(5) 将数据分成训练数据集和测试数据集：

```
    # 分成训练数据集（80%）和测试数据集（20%）
    threshold_factor = 0.8
    threshold_positive = int(threshold_factor * len(features_positive))
    threshold_negative = int(threshold_factor * len(features_negative))
```

(6) 提取特征：

```
    features_train = features_positive[:threshold_positive] +
features_negative[:threshold_negative]
    features_test = features_positive[threshold_positive:] +
features_negative[threshold_negative:]
    print "\nNumber of training datapoints:", len(features_train)
    print "Number of test datapoints:", len(features_test)
```

(7) 我们将用到朴素贝叶斯分类器。定义该对象并训练：

```
    # 训练朴素贝叶斯分类器
    classifier = NaiveBayesClassifier.train(features_train)
    print "\nAccuracy of the classifier:", nltk.classify.util.accuracy(classifier,
features_test)
```

(8) 该分类器对象包含分析过程中获得的最有信息量的单词。通过这些单词可以判定哪些可以被归类为积极评论，哪些可以被归类为消极评论。将其打印出来：

```
print "\nTop 10 most informative words:"
for item in classifier.most_informative_features()[:10]:
    print item[0]
```

(9) 生成一些随机输入句子：

```
# 输入一些简单的评论
input_reviews = [
    "It is an amazing movie",
    "This is a dull movie. I would never recommend it to anyone.",
    "The cinematography is pretty great in this movie",
    "The direction was terrible and the story was all over the place"
]
```

(10) 在这些输入句子上运行分类器，获得预测结果：

```
print "\nPredictions:"
for review in input_reviews:
    print "\nReview:", review
    probdist = classifier.prob_classify(extract_features(review.split()))
    pred_sentiment = probdist.max()
```

(11) 打印输出：

```
    print "Predicted sentiment:", pred_sentiment
    print "Probability:", round(probdist.prob(pred_sentiment), 2)
```


(12) 全部代码已经在sentiment_analysis.py文件中给出。运行该代码，可以看到命令行工具中的输出结果包括3个部分。第一个部分是准确度，如图6-8所示。

```
Number of training datapoints: 1600
Number of test datapoints: 400

Accuracy of the classifier: 0.735
```

图　6-8

(13) 第二部分输出最有信息量的单词，如图6-9所示。

```
Top 10 most informative words:
outstanding
insulting
vulnerable
ludicrous
uninvolving
astounding
avoids
fascination
animators
affecting
```

图　6-9

(14) 第三部分是对输入句子的预测列表，如图6-10所示。

```
Predictions:

Review: It is an amazing movie
Predicted sentiment: Positive
Probability: 0.61

Review: This is a dull movie. I would never recommend it to anyone.
Predicted sentiment: Negative
Probability: 0.77

Review: The cinematography is pretty great in this movie
Predicted sentiment: Positive
Probability: 0.67

Review: The direction was terrible and the story was all over the place
Predicted sentiment: Negative
Probability: 0.63
```

图　6-10

6.9.2　工作原理

这个例子将用NLTK的朴素贝叶斯分类器进行分类。在特征提取函数中，我们基本上提取了所有的唯一单词。然而，NLTK分类器需要的数据是用字典的格式存放的，因此这里用到了字典格式，便于NLTK分类器对象读取该数据。

将数据分成训练数据集和测试数据集后，可以训练该分类器，以便将句子分为积极和消极。如果查看最有信息量的那些单词，可以看到例如单词“outstanding”表示积极评论，而“insulting”表示消极评论。这是非常有趣的信息，因为它告诉我们单词可以用来表示情绪。

6.10　用主题建模识别文本的模式

主题建模是指识别文本数据隐藏模式的过程，其目的是发现一组文档的隐藏主题结构。主题建模可以更好地组织文档，以便对这些文档进行分析。主题建模是NLP研究的一个活跃领域，你可以在http://www.cs.columbia.edu/~blei/topicmodeling.html查看更多介绍。本节将用到gensim库，在进行接下来的学习之前，请确保你已经安装了该库，具体安装步骤可以参考https://radimrehurek.com/gensim/install.html。

6.10.1　详细步骤

(1) 创建一个Python文件，导入以下程序包：

```
from nltk.tokenize import RegexpTokenizer
from nltk.stem.snowball import SnowballStemmer
```

```
from gensim import models, corpora
from nltk.corpus import stopwords
```

(2) 定义一个函数来加载输入数据，本例将用到本书提供的data_topic_modeling.txt文本文件：

```
# 加载输入数据
def load_data(input_file):
    data = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data.append(line[:-1])

    return data
```

(3) 定义一个预处理文本的类。这个预处理器处理相应的对象，并从输入文本中提取相关的特征：

```
# 类预处理文本
class Preprocessor(object):
    # 对各种操作进行初始化
    def __init__(self):
        # 创建正则表达式解析器
        self.tokenizer = RegexpTokenizer(r'\w+')
```


(4) 我们需要一个停用词列表，在分析过程中可以将这些停用词排除。这些停用词都是常用词，例如“in”“the”“is”等：

```
        # 获取停用词列表
        self.stop_words_english = stopwords.words('english')
```

(5) 定义一个Snowball词干提取器：

```
        # 创建Snowball词干提取器
        self.stemmer = SnowballStemmer('english')
```

(6) 定义一个处理函数，负责标记解析、停用词去除和词干还原：

```
    # 标记解析、移除停用词、词干提取
    def process(self, input_text):
        # 标记解析
        tokens = self.tokenizer.tokenize(input_text.lower())
```

(7) 从文本中去除停用词：

```
        # 移除停用词
        tokens_stopwords = [x for x in tokens if not x in self.stop_words_english]
```

(8) 对标记做词干提取：

```
        # 词干提取
        tokens_stemmed = [self.stemmer.stem(x) for x in tokens_stopwords]
```

(9) 返回处理后的标记：

```
        return tokens_stemmed
```

(10) 定义一个main函数。从文本文件中加载输入数据：

```
if __name__=='__main__':
    # 包含输入数据的文件
    input_file = 'data_topic_modeling.txt'

    # 加载数据
    data = load_data(input_file)
```

(11) 用我们定义的类定义一个对象：

```
    # 创建预处理对象
    preprocessor = Preprocessor()
```

(12) 处理文件中的文本，并提取处理好的标记：

```
    # 创建一组经过预处理的文档
    processed_tokens = [preprocessor.process(x) for x in data]
```

(13) 创建一个基于标记文档的词典，用于主题建模：

```
    # 创建基于标记文档的词典
    dict_tokens = corpora.Dictionary(processed_tokens)
```

(14) 用处理后的标记创建一个文档-词矩阵：

```
    # 创建文档-词矩阵
    corpus = [dict_tokens.doc2bow(text) for text in processed_tokens]
```

(15) 假定文本可以分成两个主题。我们将用**隐含狄利克雷分布**（Latent Dirichlet Allocation，LDA）做主题建模。定义相关参数并初始化LDA模型对象：

```
    # 基于刚刚创建的语料库生成LDA模型
    num_topics = 2
    num_words = 4

    ldamodel = models.ldamodel.LdaModel(corpus,
            num_topics=num_topics, id2word=dict_tokens, passes=25)
```

(16) 识别出两个主题后，可以看到它是如何将两个主题分开来看的：

```
    print "\nMost contributing words to the topics:"
    for item in ldamodel.print_topics(num_topics=num_topics, num_words=num_words):
        print "\nTopic", item[0], "==>", item[1]
```

(17) 全部代码已经在topic_modeling.py文件中给出。运行该代码，可以看到命令行工具中的输出结果如图6-11所示。

```
Most contributing words to the topics:

Topic 0 ==> 0.049*need + 0.030*younger + 0.030*talent + 0.030*train

Topic 1 ==> 0.064*need + 0.063*order + 0.038*encrypt + 0.038*understand
```

图 6-11

6.10.2 工作原理

主题建模通过识别文档中最有意义、最能表征主题的词来实现主题分类。这些单词往往可以确定主题的内容。之所以使用正则表达式（regular expression）标记器，是因为只需要那些没有标点或其他标记的单词。停用词去除是另一个非常重要的步骤，因为停用词去除可以减小一些常用词（例如“is”和“the”）的噪声干扰。之后，需要对单词做词干提取，以获得其原形。以上所有步骤均被打包在一个文本分析工具的预处理模块中。在本例的代码中就是这样实现的！

本例用到了隐含狄利克雷分布技术来构建主题模型。隐含狄利克雷分布将文档表示成不同主题的混合，这些主题可以“吐出”单词。这些“吐出”的单词是有一定的概率的。隐含狄利克雷分布的目标是找到这些主题。隐含狄利克雷分布是一个生成主题的模型，该模型试图找到所有主题，而所有主题又负责生成给定主题的文档。你可以在http://blog.echen.me/2011/08/22/introduction-to-latent-dirichlet-allocation中找到更多详细介绍。

你可以在输出结果中看到，诸如单词“talent”和“train”表示运动主题，而单词“encrypt”表示密码主题。这里仅仅试验了非常小的文本文件，在这样小的文本文件中，有一些单词可能看起来不那么相关。显然，如果你用更大的数据集来运行该程序，精确度会更高。

第 7 章

语音识别

在这一章，我们将介绍以下主题：

- 读取和绘制音频数据
- 将音频信号转换为频域
- 自定义参数生成音频信号
- 合成音乐
- 提取频域特征
- 创建隐马尔科夫模型
- 创建一个语音识别器

7.1 简介

语音识别是指识别和理解口语的过程。输入音频数据，语音识别器将处理这些数据，从中提取出有用的信息。语音识别有很多实际的应用，例如声音控制设备、将语音转换成单词、安全系统等。

自然中的声音信号多种多样。同一种语言中也有很多不同的语音。语音中有很多不同的元素，例如语言、情绪、语调、噪声、口音等，我们很难定义一组构成语音的规则。尽管语音有这么多变量，人类仍然可以很轻松地理解这些。现在，我们希望机器也能以同样的方式理解语音。

在过去的几十年里，研究者们研究了语音的各个方面，例如识别说话者、理解单词、识别口音、翻译语音等。在所有的这些任务中，自动语音识别成为很多研究者重点关注的方向。在这一章中，我们将学习如何构建一个**语音识别器**。

7.2 读取和绘制音频数据

本节将介绍如何读取音频文件并将该信号进行可视化展现。这是一个好的开始，可以让我们很好地理解音频信号的基本结构。在开始之前，需要理解音频文件是实际音频信号的数字化形式，

实际的音频信号是复杂的连续波形。为了将其保存成数字化形式，需要对音频信号进行采样并将其转换成数字。例如，语音通常以44100 Hz的频率进行采样，这就意味着每秒钟信号被分解成44 100份，然后这些抽样值被保存。换句话说，每隔1/44100 s都会存储一次值。如果采样率很高，用媒体播放器收听音频时，会感觉到信号是连续的。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

(2) 使用wavfile包从input_read.wav中读取音频文件：

```
# 读取输入文件
sampling_freq, audio = wavfile.read('input_read.wav')
```

(3) 打印这个信号的相关参数：

```
# 打印参数
print '\nShape:', audio.shape
print 'Datatype:', audio.dtype
print 'Duration:', round(audio.shape[0] / float(sampling_freq), 3), 'seconds'
```

(4) 该音频信号被存储在一个16位有符号整型数据中。标准化这些值：

```
# 标准化数值
audio = audio / (2.**15)
```

(5) 提取前30个值，并将其画出：

```
# 提取前30个值画图
audio = audio[:30]
```

(6) X轴为时间轴。创建这个轴，并且X轴应该按照采样频率因子进行缩放：

```
# 建立时间轴
x_values = np.arange(0, len(audio), 1) / float(sampling_freq)
```

(7) 将单位转换为秒：

```
# 将单位转换为秒
x_values *= 1000
```

(8) 将其画出：

```
# 画出声音信号图形
plt.plot(x_values, audio, color='black')
plt.xlabel('Time (ms)')
```

```
plt.ylabel('Amplitude')
plt.title('Audio signal')
plt.show()
```

(9) 全部代码已经包含在read_plot.py文件中。运行该代码，可以看到如图7-1所示的信号。

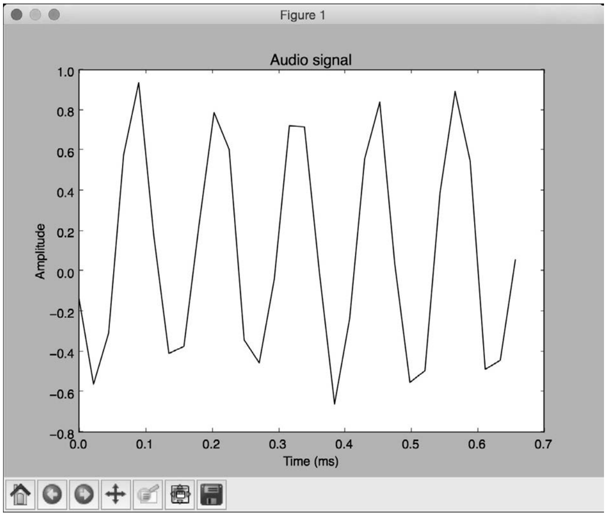


图　7-1

(10) 可以看到终端打印出如图7-2所示的结果。

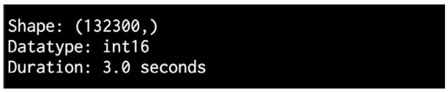


图　7-2

7.3　将音频信号转换为频域

音频信号是不同频率、幅度和相位的正弦波的复杂混合。正弦波也称作**正弦曲线**。音频信号的频率内容中隐藏了很多信息。事实上，一个音频信号的性质由其频率内容决定。世界上的语音和音乐都是基于这个事实的。在进行接下来的学习之前，你需要了解一些**傅里叶变换**（Fourier transforms）的知识，可以在http://www.thefouriertransform.com中找到快速入门介绍。下面来学习如何将音频信号转换为频域。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
from scipy.io import wavfile
import matplotlib.pyplot as plt
```

(2) 读取input_freq.wav音频文件：

```
# 读取音频文件
sampling_freq, audio = wavfile.read('input_freq.wav')
```

(3) 对信号进行标准化：

```
#  对信号进行标准化
audio = audio / (2.**15)
```

(4) 音频信号就是一个NumPy数组，因此用以下代码提取其长度：

```
# 提取数组长度
len_audio = len(audio)
```

(5) 接下来做傅里叶变换。傅里叶变换是关于中心点对称的，因此只需要转换信号的前半部分。我们的最终目标是提取功率信号，因此需要先将信号的值平方：

```
# 应用傅里叶变换
transformed_signal = np.fft.fft(audio)
half_length = np.ceil((len_audio + 1) / 2.0)
transformed_signal = abs(transformed_signal[0:half_length])
transformed_signal /= float(len_audio)
transformed_signal **= 2
```


(6) 提取信号的长度：

```
# 提取转换信号的长度
len_ts = len(transformed_signal)
```

(7) 根据信号的长度将信号乘以2：

```
# 将部分信号乘以2
if len_audio % 2:
    transformed_signal[1:len_ts] *= 2
else:
    transformed_signal[1:len_ts-1] *= 2
```

(8) 功率信号用下面的公式获得：

```
# 获取功率信号
power = 10 * np.log10(transformed_signal)
```

(9) X轴是时间轴。接下来需要根据采样频率对其进行缩放，并将其转换成秒：

```
# 建立时间轴
x_values = np.arange(0, half_length, 1) * (sampling_freq / len_ audio) / 1000.0
```

(10) 绘制该信号：

```
# 画图
plt.figure()
plt.plot(x_values, power, color='black')
plt.xlabel('Freq (in kHz)')
plt.ylabel('Power (in dB)')
plt.show()
```

(11) 全部代码已经包含在freq_transform.py文件中。运行该代码，可以看到如图7-3所示的图像。

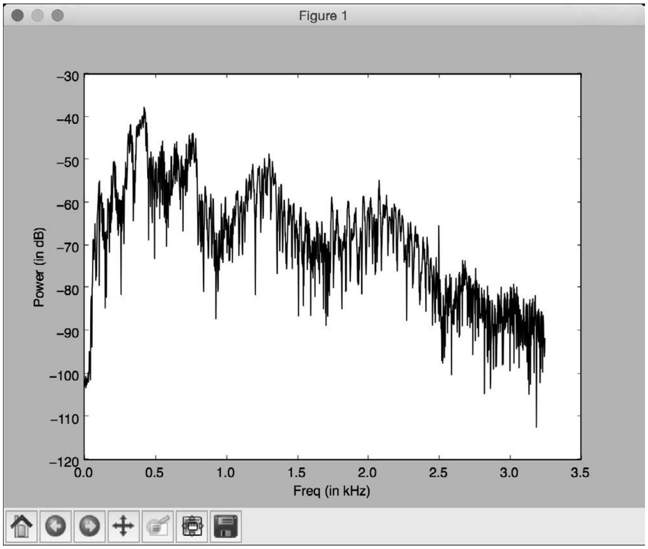


图　7-3

7.4　自定义参数生成音频信号

我们可以用NumPy生成音频信号。前面已经说过，音频信号是一些正弦波的复杂混合，下面用该原理生成自己的音频信号。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
```

(2) 定义一个输出文件，用于存储生成的音频：

```
# 定义存储音频的输出文件
output_file = 'output_generated.wav'
```

(3) 指定音频生成参数。我们希望生成一个3 s长度的信号，采样频率为44100 Hz，音频的频率为587 Hz。时间轴上的值将从$-2\times pi$到$2\times pi$：

```
# 指定音频生成的参数
duration = 3    # 单位秒
sampling_freq = 44100   # 单位Hz
tone_freq = 587
min_val = -2 * np.pi
max_val = 2 * np.pi
```

(4) 生成时间轴和音频信号。音频信号是一个简单的正弦函数，其相关参数之前已做定义：

```
# 生成音频信号
t = np.linspace(min_val, max_val, duration * sampling_freq)
audio = np.sin(2 * np.pi * tone_freq * t)
```

(5) 为信号增加一些噪声：

```
# 增加噪声
noise = 0.4 * np.random.rand(duration * sampling_freq)
audio += noise
```

(6) 将这些值转换为16位整型数，然后将其保存：

```
# 转换为16位整型数
scaling_factor = pow(2,15) - 1
audio_normalized = audio / np.max(np.abs(audio))
audio_scaled = np.int16(audio_normalized * scaling_factor)
```

(7) 将信号写入输出文件：

```
# 写入输出文件
write(output_file, sampling_freq, audio_scaled)
```

(8) 用前100个值画出该信号：

```
# 提取前100个值
audio = audio[:100]
```

(9) 生成时间轴：

```
# 生成时间轴
x_values = np.arange(0, len(audio), 1) / float(sampling_freq)
```

(10) 将时间轴的单位转换为秒：

```
# 将时间轴的单位转换为秒
x_values *= 1000
```

(11) 画出该信号：

```
# 画出音频信号图
plt.plot(x_values, audio, color='black')
```

7

```
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('Audio signal')
plt.show()
```

(12) 全部代码已经包含在generate.py文件中。运行该代码，可以看到如图7-4所示的图像。

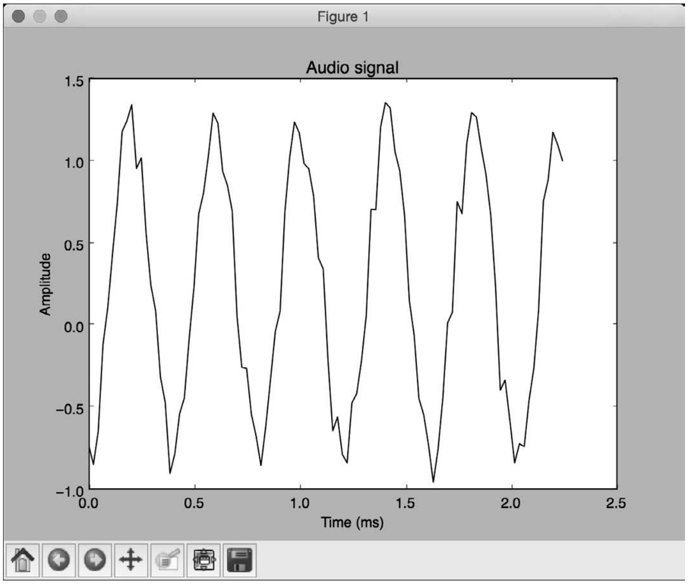


图　7-4

7.5　合成音乐

现在知道了如何生成音频，下面用同样的原理来合成一些音乐。http://www.phy.mtu.edu/~suits/notefreqs.html列举了各种音阶，例如*A*、*G*、*D*等，以及它们相应的频率。下面将用它合成简单的音乐。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import json
import numpy as np
from scipy.io.wavfile import write
import matplotlib.pyplot as plt
```

(2) 定义一个函数，该函数基于输入参数合成音调：

```
# 定义合成音调
def synthesizer(freq, duration, amp=1.0, sampling_freq=44100):
```

(3) 创建时间轴：

```
    # 创建时间轴
    t = np.linspace(0, duration, duration * sampling_freq)
```

(4) 用输入参数构建音频示例，如幅度和频率：

```
    # 构建音频信号
    audio = amp * np.sin(2 * np.pi * freq * t)

    return audio.astype(np.int16)
```

(5) 定义main函数。我们提供了文件名为tone_freq_map.json的JSON文件，该文件包括一些音阶以及它们的频率：

```
if __name__=='__main__':
    tone_map_file = 'tone_freq_map.json'
```

(6) 加载该文件：

```
    #读取频率映射文件
    with open(tone_map_file, 'r') as f:
        tone_freq_map = json.loads(f.read())
```

(7) 假设想生成2秒的*G*调：

```
    # 设置生成G调的输入参数
    tone input_tone = 'G'
    duration = 2     # 单位秒
    amplitude = 10000
    sampling_freq = 44100    # 单位Hz
```

(8) 用以下参数调用该函数：

```
    # 生成音阶
    synthesized_tone = synthesizer(tone_freq_map[input_tone], duration, amplitude,
sampling_freq)
```

(9) 将生成信号写入输出文件：

```
    # 写入输出文件
    write('output_tone.wav', sampling_freq, synthesized_tone)
```

(10) 用媒体播放器打开文件并试听，它确实是*G*调。下面做一些更有趣的事情。生成一系列的音阶，让其有一些音乐的感觉。定义一个音阶及其持续时间（秒）的序列：

```
    # 音阶及其持续时间
    tone_seq = [('D', 0.3), ('G', 0.6), ('C', 0.5), ('A', 0.3), ('Asharp', 0.7)]
```

7

(11) 迭代该序列并为它们调用合成器函数：

```
    # 构建基于和弦序列的音频信号
    output = np.array([])
    for item in tone_seq:
        input_tone = item[0]
        duration = item[1]
        synthesized_tone = synthesizer(tone_freq_map[input_tone], duration,
amplitude, sampling_freq)
        output = np.append(output, synthesized_tone, axis=0)
```

(12) 将生成信号写入输出文件：

```
    # 写入输出文件
    write('output_tone_seq.wav', sampling_freq, output)
```

(13) 全部代码已经包含在synthesize_music.py文件中。用媒体播放器打开output_tone_seq.wav文件并试听，你就可以感受到音乐！

7.6　提取频域特征

前面讨论了如何将信号转换为频域。在多数的现代语音识别系统中，人们都会用到频域特征。将信号转换为频域之后，还需要将其转换成有用的形式。**梅尔频率倒谱系数**（Mel Frequency Cepstrum Coefficient，MFCC）可以解决这个问题。MFCC首先计算信号的功率谱，然后用滤波器组和离散余弦变换的组合来提取特征。如果需要快速入门，可以查看http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs。在进行接下来的学习之前，请确保你已经安装了python_speech_features包，安装指南可以参考http://python-speech-features.readthedocs.org/en/latest。接下来介绍如何提取MFCC特征。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from features import mfcc, logfbank
```

(2) 读取input_freq.wav输入文件：

```
# 读取输入音频文件
sampling_freq, audio = wavfile.read("input_freq.wav")
```

(3) 提取MFCC和过滤器组特征：

```
# 提取MFCC和过滤器组特征
mfcc_features = mfcc(audio, sampling_freq)
filterbank_features = logfbank(audio, sampling_freq)
```

(4) 打印参数，查看可生成多少个窗体：

```
# 打印参数
print '\nMFCC:\nNumber of windows =', mfcc_features.shape[0]
print 'Length of each feature =', mfcc_features.shape[1]
print '\nFilter bank:\nNumber of windows =', filterbank_features. shape[0]
print 'Length of each feature =', filterbank_features.shape[1]
```

(5) 将MFCC特征可视化。转换矩阵，使得时域是水平的：

```
# 画出特征图
mfcc_features = mfcc_features.T
plt.matshow(mfcc_features)
plt.title('MFCC')
```

(6) 将滤波器组特征可视化。这里也需要转换矩阵，使得时域是水平的：

```
filterbank_features = filterbank_features.T
plt.matshow(filterbank_features)
plt.title('Filter bank')

plt.show()
```

(7) 全部代码已经包含在extract_freq_features.py文件中。运行该代码，可以得到如图7-5所示的MFCC特征图像。

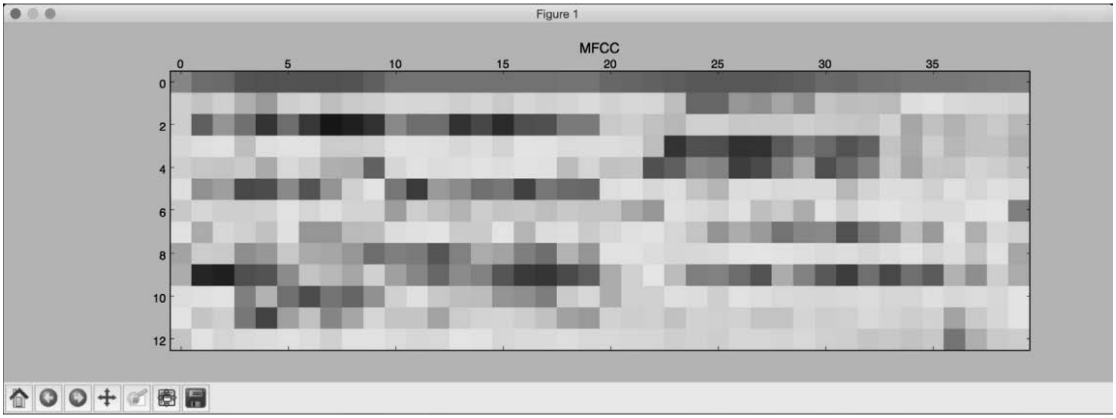


图 7-5

(8) 滤波器组特征图像如图7-6所示。

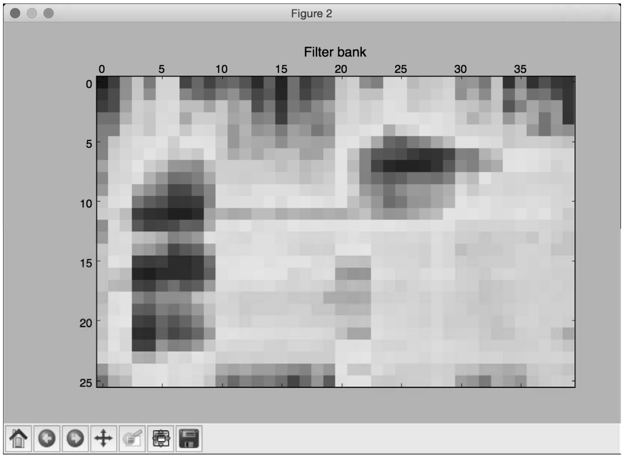


图　7-6

(9) 终端将输出如图7-7所示的结果。

```
MFCC:
Number of windows = 40
Length of each feature = 13

Filter bank:
Number of windows = 40
Length of each feature = 26
```

图　7-7

7.7　创建隐马尔科夫模型

接下来可以讨论语音识别了，本例将用到**隐马尔科夫模型**（Hidden Markov Models，HMMs）来做语音识别。隐马尔科夫模型非常擅长建立时间序列数据模型。因为一个音频信号同时也是一个时间序列信号，因此隐马尔科夫模型也同样适用于音频信号的处理。假定输出是通过隐藏状态生成的，我们的目标是找到这些隐藏状态，以便对信号建模。你可以在https://www.robots.ox.ac.uk/~vgg/rg/slides/hmm.pdf查看更多关于隐马尔科夫模型的介绍。进行接下来的学习之前，请确保你已经安装了hmmlearn包，安装说明可查看http://hmmlearn.readthedocs. org/en/latest。接下来学习如何创建隐马尔科夫模型。

详细步骤

(1) 创建一个Python文件，定义一个类来创建隐马尔科夫模型：

```
# 创建类处理HMM相关过程
class HMMTrainer(object):
```

(2) 初始化该类。下面将用到高斯隐马尔科夫模型（Gaussian HMMs）来对数据建模。参数n_components定义了隐藏状态的个数，参数cov_type定义了转移矩阵的协方差类型，参数n_iter定义了训练的迭代次数：

```
    def __init__(self, model_name='GaussianHMM', n_components=4, cov_type='diag',
n_iter=1000):
```

这些参数的选定取决于你的需求。只有正确地理解这些参数的含义，才能灵活地进行运用。

(3) 初始化变量：

```
        self.model_name = model_name
        self.n_components = n_components
        self.cov_type = cov_type
        self.n_iter = n_iter
        self.models = []
```

(4) 用以下参数定义模型：

```
        if self.model_name == 'GaussianHMM':
            self.model = hmm.GaussianHMM(n_components=self.n_components,
                    covariance_type=self.cov_type, n_iter=self.n_iter)
        else:
            raise TypeError('Invalid model type')
```

(5) 输入数据是一个NumPy数组，数组的每个元素都是一个特征向量，每个特征向量都包含*k*个维度：

```
    # X是二维数组，其中每一行是13维
    def train(self, X):
        np.seterr(all='ignore')
        self.models.append(self.model.fit(X))
```

(6) 基于该模型定义一个提取分数的方法：

```
    # 对输入数据运行模型
    def get_score(self, input_data):
        return self.model.score(input_data)
```

(7) 前面创建了一个类来处理隐马尔科夫模型的训练和预测，但是还需要一些数据来观察它的运行情况。下一节将创建一个语音识别器，全部代码包含在speech_recognizer.py文件中。

7.8　创建一个语音识别器

本节需要一个语音文件数据库来创建语音识别器，用到的数据库文件保存在https://code.google.com/archive/p/hmm-speech-recognition/downloads中。其中包含7个不同的单词，并且每个单词都有15个音频文件与之相关。这是一个较小的数据集，但是足够我们理解如何创建一个语音识别器并识别7个不同的单词。我们需要为每一类构建一个隐马尔科夫模型。如果想识别新的输入

文件中的单词，需要对该文件运行所有的模型，并找出最佳分数的结果。下面将用到在前一节构建的隐马尔科夫类。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import os
import argparse

import numpy as np
from scipy.io import wavfile
from hmmlearn import hmm
from features import mfcc
```

(2) 定义一个函数来解析命令行中的输入参数：

```
# 解析输入参数的函数
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the HMM classifier')
    parser.add_argument("--input-folder", dest="input_folder", required=True,
            help="Input folder containing the audio files insubfolders")
    return parser
```

(3) 定义main函数，解析输入参数：

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_folder = args.input_folder
```

(4) 初始化隐马尔科夫模型的变量：

```
    hmm_models = []
```

(5) 解析包含所有数据库音频文件的输入路径：

```
    # 解析输入路径
    for dirname in os.listdir(input_folder):
```

(6) 提取子文件夹的名称：

```
        # 获取子文件夹名称
        subfolder = os.path.join(input_folder, dirname)

        if not os.path.isdir(subfolder):
            continue
```

(7) 子文件夹的名称即为该类的标记。用以下方式将其提取出来：

```
        # 提取标记
        label = subfolder[subfolder.rfind('/') + 1:]
```

(8) 初始化用于训练的变量：

```
# 初始化变量
X = np.array([])
y_words = []
```

(9) 迭代每一个子文件夹中的音频文件：

```
# 迭代所有音频文件（分别保留一个进行测试）
for filename in [x for x in os.listdir(subfolder) if x.endswith('.wav')][:-1]:
```

(10) 读取每个音频文件：

```
    # 读取每个音频文件
    filepath = os.path.join(subfolder, filename)
    sampling_freq, audio = wavfile.read(filepath)
```

(11) 提取MFCC特征：

```
    # 提取MFCC特征
    mfcc_features = mfcc(audio, sampling_freq)
```

(12) 将MFCC特征添加到X变量：

```
    # 将MFCC特征添加到X变量
    if len(X) == 0:
        X = mfcc_features
    else:
        X = np.append(X, mfcc_features, axis=0)
```


(13) 同时添加标记信息：

```
    # 添加标记
    y_words.append(label)
```

(14) 一旦提取完当前类所有文件的特征，就可以训练并保存隐马尔科夫模型了。因为隐马尔科夫模型是一个无监督学习的生成模型，所以并不需要利用标记针对每一类构建隐马尔科夫模型。假定每个类都将构建一个隐马尔科夫模型：

```
# 训练并保存HMM模型
hmm_trainer = HMMTrainer()
hmm_trainer.train(X)
hmm_models.append((hmm_trainer, label))
hmm_trainer = None
```

(15) 获取一个未被用于训练的测试文件列表：

```
# 测试文件
input_files = [
        'data/pineapple/pineapple15.wav',
        'data/orange/orange15.wav',
        'data/apple/apple15.wav',
        'data/kiwi/kiwi15.wav'
        ]
```

(16) 解析输入文件：

```
#为输入数据分类
for input_file in input_files:
```

(17) 读取每个音频文件：

```
    # 读取每个音频文件
    sampling_freq, audio = wavfile.read(input_file)
```

(18) 提取MFCC特征：

```
    # 提取MFCC特征
    mfcc_features = mfcc(audio, sampling_freq)
```

(19) 定义两个变量，分别用于存放最大分数值和输出标记：

```
    # 定义变量
    max_score = None
    output_label = None
```

(20) 迭代所有模型，并通过每个模型运行输入文件：

```
    # 迭代HMM模型并选取得分最高的模型
    for item in hmm_models:
        hmm_model, label = item
```

(21) 提取分数，并保存最大分数值：

```
        score = hmm_model.get_score(mfcc_features)
        if score > max_score:
            max_score = score
            output_label = label
```

(22) 打印真实的、预测的标记：

```
    # 打印结果
    print "\nTrue:", input_file[input_file.find('/')+1:input_ file.rfind('/')]
    print "Predicted:", output_label
```

(23) 全部代码已经包含在speech_recognizer.py文件中。运行该代码，可以在终端看到如图7-8所示的显示结果。



图　7-8

第 8 章

解剖时间序列和时序数据

在这一章，我们将介绍以下主题：

- 将数据转换为时间序列格式
- 切分时间序列数据
- 操作时间序列数据
- 从时间序列数据中提取统计数字
- 针对序列数据创建隐马尔科夫模型
- 针对序列文本数据创建条件随机场
- 用隐马尔科夫模型分析股票市场数据

8.1 简介

时间序列数据就是随着时间的变化收集的测量序列数据。这些数据是根据预定义的变量并在固定的间隔时间采集的。时间序列数据最主要的特征就是其顺序是非常关键的。

我们收集的数据是按照时间轴排序的，它们的出现顺序包含很多隐藏的模式和信息。如果改变顺序，则将彻底改变数据的含义。序列数据的广义概念是指任意序列形式的数据，包括时间序列数据。

我们的目标是构建一个模型，该模型描述了时间序列或任意序列的模式，用于描述时间序列模式的重要特征。可以用这些模型解释过去可能会影响到未来，查看两个数据集是如何相互关联的，如何预测未来可能的值，或者如何控制基于某个度量标准的给定变量。

为了将时间序列数据可视化，我们倾向于将其用折线图或柱状图画出。时间序列数据分析常用于金融、信号处理、天气预测、轨道预测、地震预测或者任意需要处理时间数据的场合。我们在时间序列和顺序数据分析中构建的模型应该考虑数据的顺序，并提取相互之间的关系。接下来分析Python中的时间序列和顺序数据。

8.2　将数据转换为时间序列格式

下面以理解如何将一系列观察结果转换为时间序列数据并将其可视化作为本章的开始。本例将用到pandas库来分析时间序列数据。在进行接下来的学习之前，请确保已经安装好了pandas库，安装方法可以参考http://pandas.pydata.org/pandas-docs/stable/install.htm。

详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

(2) 定义一个函数来读取输入文件，该文件将序列观察结果转换为时间序列数据：

```
def convert_data_to_timeseries(input_file, column, verbose=False):
```

(3) 这里将用到一个包含4列的文本文件，其中第一列表示年，第二列表示月，第三列和第四列表示数据。将文件加载到NumPy数组：

```
    # 加载输入文件
    data = np.loadtxt(input_file, delimiter=',')
```

(4) 因为数据是按时间的前后顺序排列的，数据的第一行是起始日期，而数据的最后一行是终止日期。下面提取出数据集的起始日期和终止日期：

```
    # 提取起始日期和终止日期
    start_date = str(int(data[0,0])) + '-' + str(int(data[0,1]))
    end_date = str(int(data[-1,0] + 1)) + '-' + str(int(data[-1,1]% 12 + 1))
```

(5) 这个函数还有一个详细的版本。因此，当这个值为真时，就打印一些信息。打印出起始日期和终止日期：

```
    if verbose:
       print "\nStart date =", start_date
       print "End date =", end_date
```

(6) 创建一个pandas变量，该变量包含了以月为间隔的日期序列：

```
    # 创建以月为间隔的变量
    dates = pd.date_range(start_date, end_date, freq='M')
```

(7) 下一步是将给定的列转换为时间序列数据。可以用年和月访问这些数据（而不是索引）：

```
    # 将日期转换成时间序列
    data_timeseries = pd.Series(data[:,column], index=dates)
```

(8) 打印出最开始的10个元素：

```
    if verbose:
        print "\nTime series data:\n", data_timeseries[:10]
```

(9) 返回时间索引变量：

```
    return data_timeseries
```

(10) 定义main函数：

```
if __name__=='__main__':
```

(11) 本例将使用本书提供的data_timeseries.txt文件：

```
    # 输入数据文件
    input_file = 'data_timeseries.txt'
```

(12) 加载文本文件的第三列，并将其转换为时间序列数据：

```
    # 加载输入数据
    column_num = 2
    data_timeseries = convert_data_to_timeseries(input_file, column_num)
```

(13) pandas库提供了非常实用的画图功能，你可以直接在变量上运行：

```
    # 画出数据序列数据
    data_timeseries.plot()
    plt.title('Input data')

    plt.show()
```

(14) 全部代码已经包含在convert_to_timeseries.py文件中。运行该代码，可以看到如图8-1所示的图像。

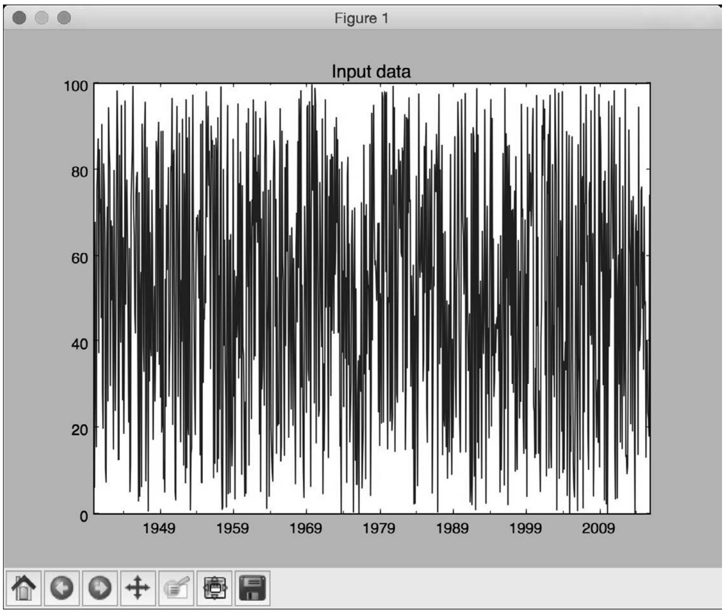


图　8-1

8.3　切分时间序列数据

这一节将介绍如何用pandas切分时间序列数据，帮助你从时间序列数据的各个阶段获得相应的信息。接下来将学习如何用日期处理数据集的子集。

详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from convert_to_timeseries import convert_data_to_timeseries
```

(2) 使用前一节中用到的文本文件，对该文本文件中的数据进行切分：

```
# 输入数据文件
input_file = 'data_timeseries.txt'
```

(3) 又一次用到第三列数据：

```
# 加载数据
column_num = 2
data_timeseries = convert_data_to_timeseries(input_file, column_ num)
```

(4) 假定我们希望提取给定的起始年份和终止年份之间的数据，下面做如下定义：

```
# 确定画图起止年份
start = '2008'
end = '2015'
```

(5) 画出给定年份范围内的数据：

```
plt.figure()
data_timeseries[start:end].plot()
plt.title('Data from ' + start + ' to ' + end)
```

(6) 还可以在给定月份范围内切分数据：

```
# 画图起止年月
start = '2007-2'
end = '2007-11'
```

(7) 画出数据：

```
plt.figure()
data_timeseries[start:end].plot()
plt.title('Data from ' + start + ' to ' + end)

plt.show()
```

(8) 全部代码已经包含在slicing_data.py文件中。运行该代码，可以看到如图8-2所示的图像。

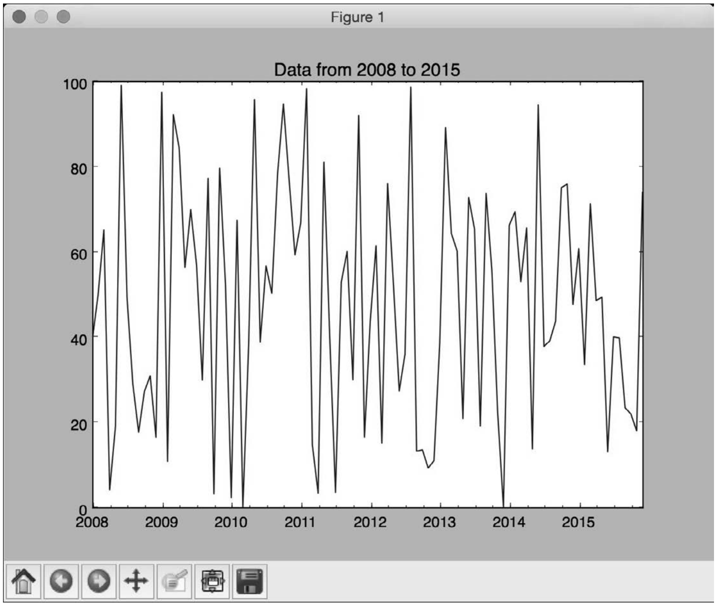


图 8-2

(9) 如图8-3所示的图像展示了更小时间范围内的数据，因此它看起来像是做了放大处理。

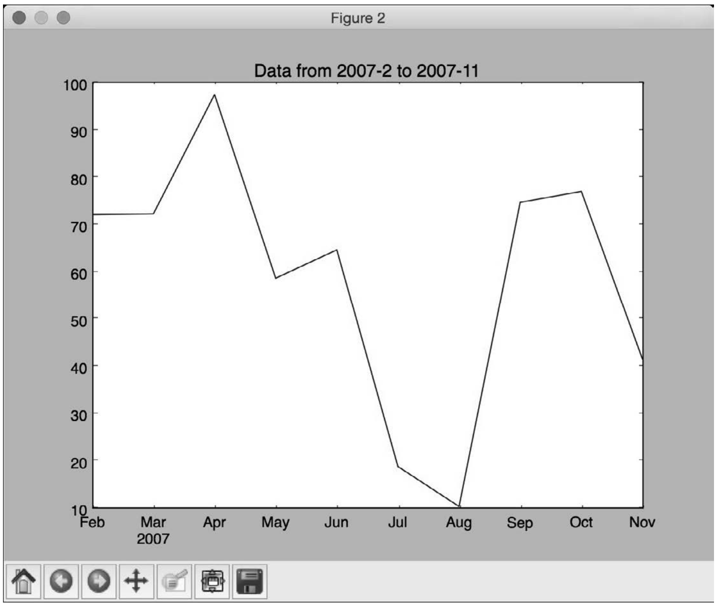


图 8-3

8.4　操作时间序列数据

现在我们知道如何切分数据并抽取各种子数据集了，接下来介绍如何操作时间序列数据。你可以用各种不同的方式过滤数据。pandas库提供了各种操作时间序列数据的方式。

详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from convert_to_timeseries import convert_data_to_timeseries
```

(2) 使用上一节用到的文本文件：

```
# 输入数据文件
input_file = 'data_timeseries.txt'
```

(3) 将用到第三列和第四列数据：

```
# 加载数据
data1 = convert_data_to_timeseries(input_file, 2)
data2 = convert_data_to_timeseries(input_file, 3)
```

(4) 将数据转化为pandas的数据帧：

```
dataframe = pd.DataFrame({'first': data1, 'second': data2})
```

(5) 画出给定年份范围内的数据：

```
# 画图
dataframe['1952':'1955'].plot()
plt.title('Data overlapped on top of each other')
```

(6) 假定我们希望画出在给定年份范围内刚才加载的两列数据的不同，可以用以下方式实现：

```
# 画出两组数据的不同
plt.figure()
difference = dataframe['1952':'1955']['first'] - dataframe['1952':'1955']['second']
difference.plot()
plt.title('Difference (first - second)')
```

(7) 如果希望对第一列和第二列用不同的条件来过滤数据，可以指定这些条件并将其画出：

```
# 当“first”大于某个阈值且“second”小于某个阈值时
dataframe[(dataframe['first'] > 60) & (dataframe['second'] < 20)]. plot()
plt.title('first > 60 and second < 20')

plt.show()
```

(8) 全部代码已经包含在operating_on_data.py文件中。运行该代码，可以看到如图8-4所示的第一幅图像。

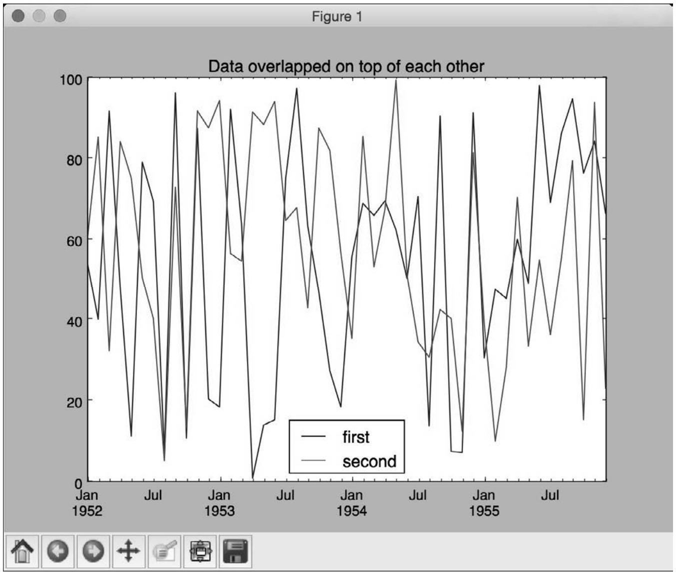


图 8-4

(9) 如图8-5所示的第二幅图像表示出不同之处。

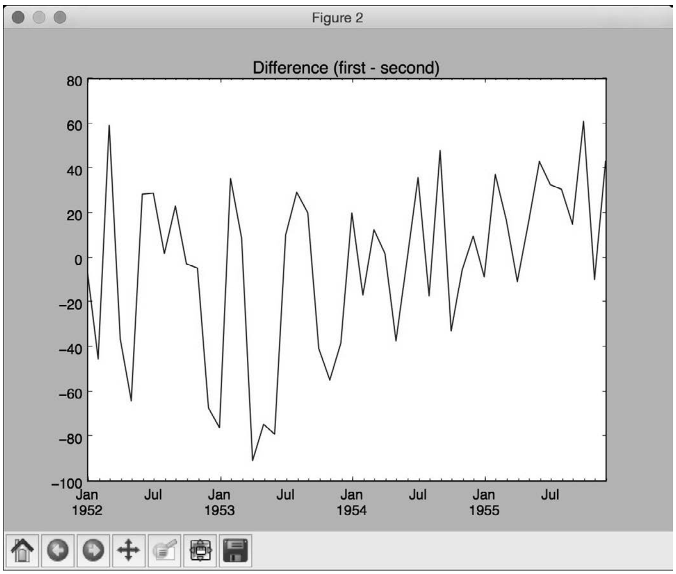


图 8-5

(10) 如图8-6所示的第三幅图像表示过滤数据。

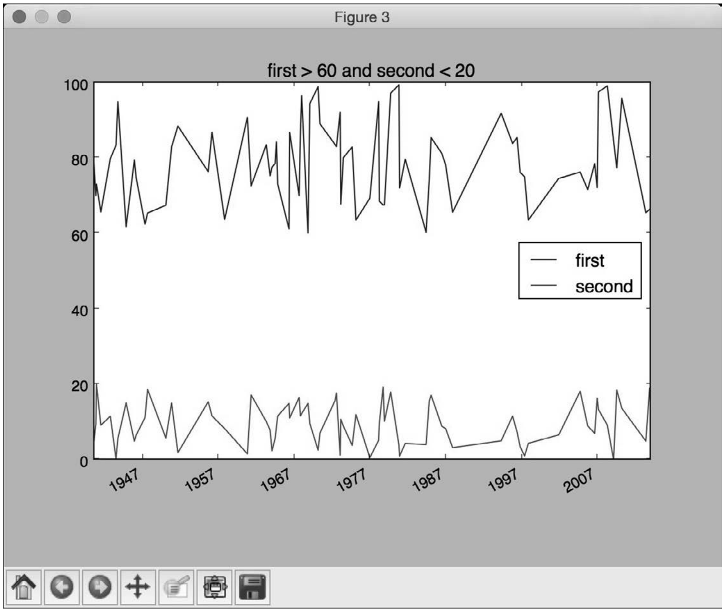


图　8-6

8.5　从时间序列数据中提取统计数字

分析时间序列数据的主要原因之一是从中提取出有趣的统计信息。考虑数据的本质，时间序列分析可以提供很多信息。本节将介绍如何提取这些统计信息。

详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from convert_to_timeseries import convert_data_to_timeseries
```

(2) 用到前一节用到的文本文件：

```
# 输入数据文件
input_file = 'data_timeseries.txt'
```

(3) 加载第三列和第四列数据：

```
# 加载数据
data1 = convert_data_to_timeseries(input_file, 2)
data2 = convert_data_to_timeseries(input_file, 3)
```

(4) 创建一个pandas数据结构来保存这些数据，这个数据看着比较像词典，它有对应的键和值：

```
dataframe = pd.DataFrame({'first': data1, 'second': data2})
```

(5) 接下来提取一些统计数据。用以下代码提取最大值和最小值：

```
#打印最大值和最小值
print '\nMaximum:\n', dataframe.max()
print '\nMinimum:\n', dataframe.min()
```

(6) 打印数据的均值或者是每行的均值：

```
# 打印均值
print '\nMean:\n', dataframe.mean()
print '\nMean row-wise:\n', dataframe.mean(1)[:10]
```

(7) 滑动均值是在时间序列分析中较常用的统计。其最著名的应用之一是平滑信号以去除噪声。滑动均值是指计算一个窗口范围内的信号均值，并不断地移动时间窗。这里用到的窗口大小为24：

```
# 打印滑动均值
pd.rolling_mean(dataframe, window=24).plot()
```

(8) 相关性系数对于理解数据的本质来说非常有用：

```
# 打印相关性系数
print '\nCorrelation coefficients:\n', dataframe.corr()
```

(9) 用大小为60的窗口将其画出：

8

```
# 画出滑动相关性
plt.figure()
pd.rolling_corr(dataframe['first'], dataframe['second'], window=60).plot()

plt.show()
```

(10) 全部代码已经包含在extract_stats.py文件中。运行该代码，可以看到画出的滑动均值如图8-7所示。

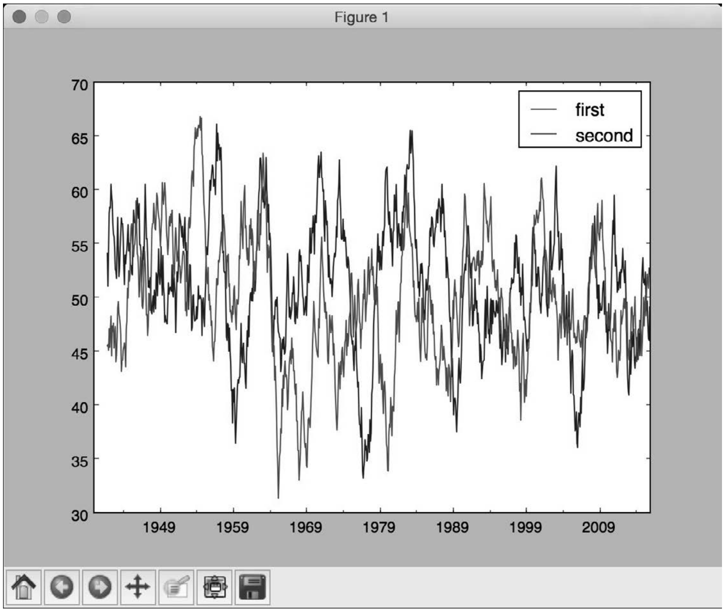


图 8-7

(11) 第二幅图像表示滑动相关性，如图8-8所示。

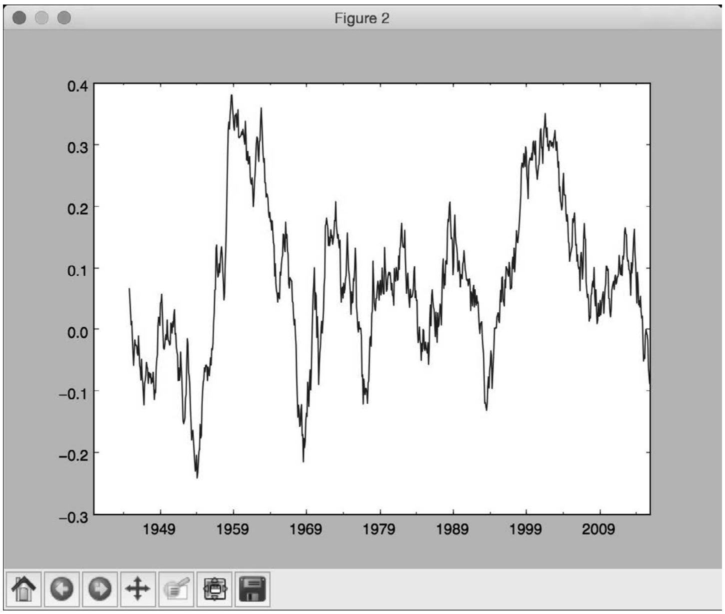


图 8-8

(12) 在命令行工具输出的上半部分，可以看到最大值、最小值和均值，如图8-9所示。

```
Maximum:
first     99.82
second    99.97
dtype: float64

Minimum:
first     0.07
second    0.00
dtype: float64

Mean:
first     51.264529
second    49.695417
dtype: float64
```

图 8-9

(13) 在命令行工具输出的下半部分，可以看到每行的均值和相关系数，如图8-10所示。

```
Mean row-wise:
1940-01-31    81.885
1940-02-29    41.135
1940-03-31    10.305
1940-04-30    83.545
1940-05-31    18.395
1940-06-30    16.695
1940-07-31    86.875
1940-08-31    42.255
1940-09-30    55.880
1940-10-31    34.720
Freq: M, dtype: float64

Correlation coefficients:
           first    second
first   1.000000  0.077607
second  0.077607  1.000000
```

图 8-10

8.6 针对序列数据创建隐马尔科夫模型

隐马尔科夫模型（HMMs）是处理序列数据非常强大的方法，广泛应用于金融、语音分析、天气预测、单词序列等领域。我们往往对发现随时间变化的隐藏模式非常感兴趣。

任何产生输出序列的数据源均可以产生模式。请注意，HMMs是一个生成模型，这也就意味着一旦掌握了其底层结构，就可以产生数据。HMMs并不能对基础形式的类进行区分，这与那些可以做类区分的判定模型形成鲜明的对比，但是这些可以做类区分的判定模型却不能生成数据。

8.6.1　准备工作

例如，我们希望预测明天的天气是晴天、阴天或下雨。为了实现预测，需要查看所有的参数，例如温度、气压等，而潜在的状态是隐藏的。这里，潜在的状态是指3个可选状态：晴天、阴天或下雨。如果希望了解更多HMMs的详细介绍，可以在https://www.robots.ox.ac.uk/~vgg/rg/slides/hmm.pdf找到相关信息。

本例将用到hmmlearn来创建和训练HMMs，在进行接下来的学习之前，请确保你已经安装了hmmlearn，安装说明请查看http://hmmlearn.readthedocs.org/en/latest。

8.6.2　详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import datetime

import numpy as np
import matplotlib.pyplot as plt
from hmmlearn.hmm import GaussianHMM

from convert_to_timeseries import convert_data_to_timeseries
```

(2) 本例将用到提供的data_hmm.txt文件。该文件包括带逗号分隔符的行。每行包括3个值：一个年份、一个月份和一个浮点型数据。下面将它加载到一个NumPy数组中：

```
# 从输入文件加载数据
input_file = 'data_hmm.txt'
data = np.loadtxt(input_file, delimiter=',')
```

(3) 将数据按照列的方向堆叠起来用于分析。我们并不需要在技术上做列堆叠，因为只有一个列，但如果你有多于一个列要进行分析，那么可以用下面的代码实现：

```
# 排列训练数据
X = np.column_stack([data[:,2]])
```

(4) 用4个成分创建并训练HMM。成分的个数是一个需要进行选择的超参数。这里选择4个成分，也就意味着用4个潜在状态生成数据。接下来看看这个参数的性能如何变化：

```
# 创建并训练高斯HMM模型
print "\nTraining HMM...."
num_components = 4
model = GaussianHMM(n_components=num_components, covariance_ type="diag",
n_iter=1000)
model.fit(X)
```

(5) 运行预测器以获得隐藏状态：

```
# 预测HMM的隐藏状态
hidden_states = model.predict(X)
```

(6) 计算这些隐藏状态的均值和方差：

```
print "\nMeans and variances of hidden states:"
for i in range(model.n_components):
    print "\nHidden state", i+1
    print "Mean =", round(model.means_[i][0], 3)
    print "Variance =", round(np.diag(model.covars_[i])[0], 3)
```

(7) 正如前面所述，HMM是一个生成模型，因此这里生成1000个示例数据并将其画出：

```
# 用模型生成数据
num_samples = 1000
samples, _ = model.sample(num_samples)
plt.plot(np.arange(num_samples), samples[:,0], c='black')
plt.title('Number of components = ' + str(num_components))

plt.show()
```

(8) 全部代码已经包含在hmm.py文件中。运行该代码，可以看到如图8-11所示的图像。

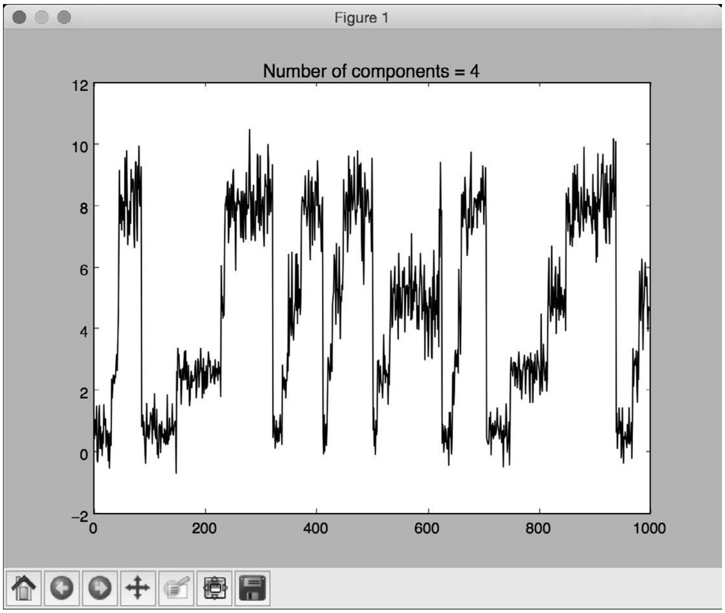


图 8-11

(9) 可以用n_components参数进行试验，看看随着其值的增加，曲线会变得更好，也可以通过设定更多的隐藏状态来训练和自定义模型。如果将这个参数设为8，其结果如图8-12所示。

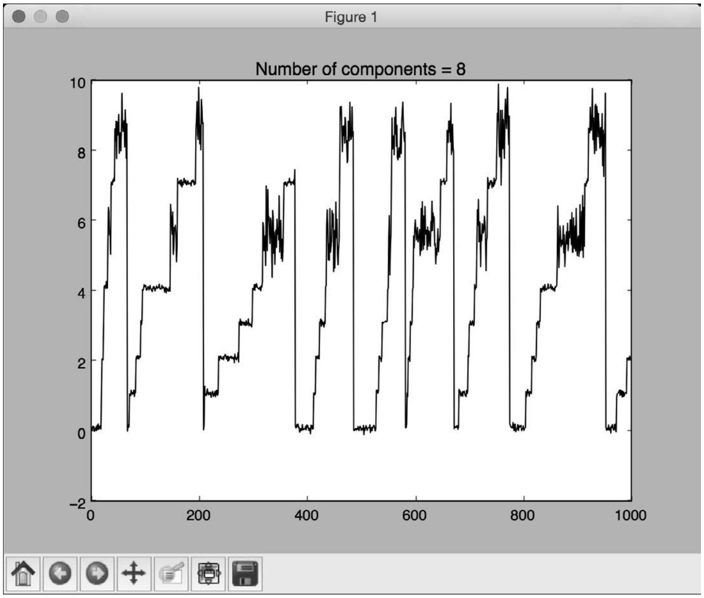


图　8-12

(10) 如果将它增加到`12`，图像会变得更平滑，如图8-13所示。

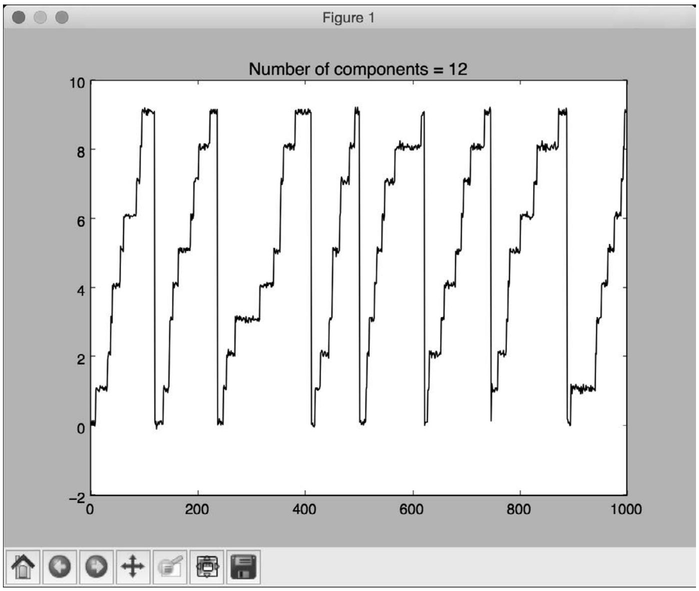


图　8-13

(11) 在命令行工具中，可以看到如图8-14所示的输出结果。

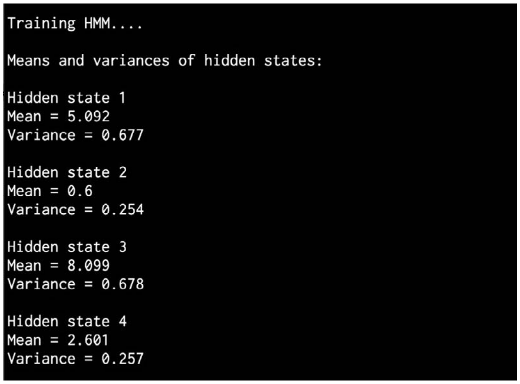


图　8-14

8.7　针对序列文本数据创建条件随机场

条件随机场（Conditional Random Fields，CRFs）是一个概率模型，该模型用于分析结构化数据。条件随机场常用于标记和分段序列数据。条件随机场与隐马尔科夫模型相反，它是一个判定模型，而隐马尔科夫模型是一个生成模型。条件随机场用于分析序列、股票、语音、单词等。在这些模型中，给定一个带标签的观察序列，对这个序列定义一个条件随机分布。这与隐马尔科夫模型相反，隐马尔科夫模型定义的是对标签和观察序列的联合分布。

8.7.1　准备工作

隐马尔科夫模型假设当前的输出是与之前的输出独立统计的。这是隐马尔科夫模型所需要的，以确保该假设能够以一种健壮的方式工作。然而，这个假设并不总是成立。时间序列中的当前输出往往取决于之前的输出。条件随机场模型优于隐马尔科夫模型的一点在于它们是由自然条件决定的，也就是说，条件随机场模型并不假设输出观察值之间的独立性。不仅如此，条件随机场还有一些优于隐马尔科夫模型的地方。条件随机场模型在诸如语言学、生物信息学、语音分析等领域的应用都优于隐马尔科夫模型。这一节将介绍如何用条件随机场模型分析字母序列。

本例将用到pystruct库来构造和训练条件随机场。在进行接下来的学习之前，请确保你已经安装了它，安装说明可查看https://pystruct.github.io/installation.html。

8.7.2　详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import os
import argparse
import cPickle as pickle
```

```
import numpy as np
import matplotlib.pyplot as plt
from pystruct.datasets import load_letters
from pystruct.models import ChainCRF
from pystruct.learners import FrankWolfeSSVM
```

(2) 定义一个参数解析器，并将C值作为输入参数。C是一个超参数，该参数控制你想要的模型的具体程度，而不会失去一般化的能力：

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the CRF classifier')
    parser.add_argument("--c-value", dest="c_value", required=False, type=float,
            default=1.0, help="The C value that will be used for training")
    return parser
```

(3) 定义一个处理所有与CRF相关处理的类：

```
class CRFTrainer(object):
```

(4) 定义一个init函数并初始化其值：

```
    def __init__(self, c_value, classifier_name='ChainCRF'):
        self.c_value = c_value
        self.classifier_name = classifier_name
```

(5) 下面将用链式条件随机场来分析数据。这里需要加上一个错误检查：

```
        if self.classifier_name == 'ChainCRF':
            model = ChainCRF()
```

(6) 定义一个在条件随机场模型中需要用到的分类器。这里用**支持向量机**（Support Vector Machine）来实现：

```
            self.clf = FrankWolfeSSVM(model=model, C=self.c_value, max_iter=50)
        else:
            raise TypeError('Invalid classifier type')
```

(7) 加载字母数据集。这个数据集包括分割的字母以及和其相关的特征向量。因为已经有了特征向量，所以不需要分析图像。每个单词的首字母都已被去掉，所以剩下的字母都是小写字母：

```
    def load_data(self):
        letters = load_letters()
```

(8) 加载数据和标签到其相应的变量：

```
        X, y, folds = letters['data'], letters['labels'], letters['folds']
        X, y = np.array(X), np.array(y)
        return X, y, folds
```

(9) 定义一个训练方法：

```
    # X是由样本组成一个numpy数组，每个样本的维度是(字母, 数值)
    def train(self, X_train, y_train):
        self.clf.fit(X_train, y_train)
```

(10) 定义一个方法来评价模型的性能：

```
    def evaluate(self, X_test, y_test):
        return self.clf.score(X_test, y_test)
```

(11) 定义一个方法，将新数据进行分类：

```
    # 对输入数据运行分类器
    def classify(self, input_data):
        return self.clf.predict(input_data)[0]
```

(12) 这些字母在编号的数组中被索引。为了检查输出并将其变得可读，需要将这些数字转换为字母。下面定义一个函数来执行这样的转换：

```
def decoder(arr):

    alphabets = 'abcdefghijklmnopqrstuvwxyz'
    output = ''
    for i in arr:
        output += alphabets[i]

    return output
```

(13) 定义main函数并解析输入参数：

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    c_value = args.c_value
```

(14) 用类和C值初始化变量：

```
    crf = CRFTrainer(c_value)
```

(15) 加载字母数据：

```
    X, y, folds = crf.load_data()
```

(16) 将数据分成训练数据集和测试数据集：

```
    X_train, X_test = X[folds == 1], X[folds != 1]
    y_train, y_test = y[folds == 1], y[folds != 1]
```

(17) 训练CRF模型：

```
    print "\nTraining the CRF model..."
    crf.train(X_train, y_train)
```

(18) 评价CRF模型的性能：

```
    score = crf.evaluate(X_test, y_test)
    print "\nAccuracy score =", str(round(score*100, 2)) + '%'
```

(19) 输入一个随机测试向量，并用这个模型预测输出：

```
    print "\nTrue label =", decoder(y_test[0])
    predicted_output = crf.classify([X_test[0]])
    print "Predicted output =", decoder(predicted_output)
```

(20) 全部代码已经包含在crf.py文件中。运行该代码，可以看到在命令行工具中显示如图8-15所示的输出结果。正如你看到的，这个单词应该是“commanding”，而条件随机场模型很好地预测了这个结果。

```
Training the CRF model...

Accuracy score = 78.05%

True label = ommanding
Predicted output = ommanging
```

图　8-15

8.8　用隐马尔科夫模型分析股票市场数据

本节将用隐马尔科夫模型来预测股票数据。股票市场数据是典型的时间序列数据示例，其数据都是用日期格式来组织的。在即将用到的数据集中，可以看到各个公司的股票数据随时间的波动情况。隐马尔科夫模型是生成模型，可用于分析这样的时间序列数据。这一节将用这些模型分析股票价格。

详细步骤

(1) 创建一个新的Python文件，并导入以下程序包：

```
import datetime

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo_ochl
from hmmlearn.hmm import GaussianHMM
```

(2) 从雅虎财经（Yahoo finance）获取股票报价。在matplotlib库中有一个函数可以直接加载：

```
# 从雅虎财经获取股票报价
quotes = quotes_historical_yahoo_ochl("INTC", datetime.date(1994, 4, 5),
         datetime.date(2015, 7, 3))
```

(3) 每个报价包含6个值。下面提取相关数据，如股票的收盘价和一定时期内股票的成交量：

```
# 提取需要的数值
dates = np.array([quote[0] for quote in quotes], dtype=np.int)
closing_values = np.array([quote[2] for quote in quotes])
volume_of_shares = np.array([quote[5] for quote in quotes])[1:]
```

(4) 计算每天收盘价的变化率，用这个变化率作为一个特征：

```
# 计算每天收盘价的变化率
diff_percentage = 100.0 * np.diff(closing_values) / closing_ values[:-1]

dates = dates[1:]
```

(5) 将两个数组进行列堆叠，以用作训练：

```
# 将变化率与交易量组合起来
X = np.column_stack([diff_percentage, volume_of_shares])
```

(6) 用5个成分训练隐马尔科夫模型：

```
# 创建并训练高斯HMM模型
print "\nTraining HMM...."
model = GaussianHMM(n_components=5, covariance_type="diag", n_ iter=1000)

model.fit(X)
```

(7) 生成500个示例数据用于训练隐马尔科夫模型，并将其画出：

```
# 用模型生成数据
num_samples = 500
samples, _ = model.sample(num_samples)
plt.plot(np.arange(num_samples), samples[:,0], c='black')

plt.show()
```

(8) 全部代码已经包含在hmm_stock.py文件中。运行该代码，可以看到如图8-16所示的图像。

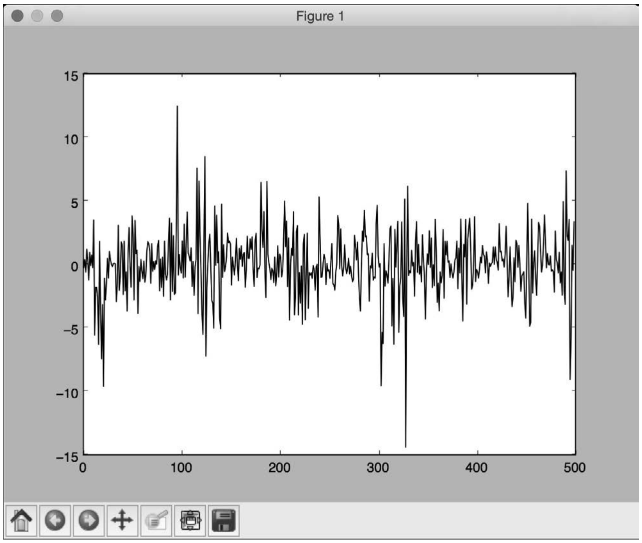


图 8-16

第 9 章

图像内容分析

在这一章，我们将介绍以下主题：

- 用OpenCV-Pyhon操作图像
- 检测边
- 直方图均衡化
- 检测棱角
- 检测SIFT特征点
- 创建Star特征检测器
- 利用视觉单词码本和向量量化创建特征
- 用极端随机森林训练图像分类器
- 创建一个对象识别器

9.1 简介

计算机视觉是一个研究如何处理、分析和理解视觉数据内容的领域。在图像内容分析中，会用到很多计算机视觉算法来构建我们对图像对象的理解。计算机视觉包括很多方面的图像分析，例如目标识别、形状分析、姿态估计、3D建模、视觉搜索等。人类非常擅长鉴定和识别其周边的事物，而计算机视觉的终极目标就是用计算机准确地模拟人类的视觉系统。

计算机视觉包括多个级别的分析。在低级视觉分析领域，计算机视觉可以进行像素处理，例如边检测、形态处理和光流。在中级和高级视觉分析领域，计算机视觉可以处理事物，例如物体识别、3D建模、运动分析以及其他方面的视觉数据。随着分析层次的深入，我们会对视觉系统的各个概念钻研得更加深入，并基于活动和意图提取出对视觉数据的描述。值得注意的一点是，高层次的分析往往依赖低层次分析的输出结果。

关于计算机视觉最常见的一个问题是“计算机视觉与图像处理有什么不同”。图像处理是在像素级别对图像进行变换。图像处理系统的输入和输出都是图像，常用的图像处理有边检测、直方图均衡化或图像压缩。计算机视觉算法大量依赖了图像处理算法来执行其任务。在计算机视觉

领域，我们还处理更复杂的事情，例如在概念层级理解视觉数据，期望借此帮助自己构建对图像对象更有意义的描述。计算机视觉系统的输出是给定图像的3D场景的描述，这样的描述可以是各种形式的，而这取决于你的需要。

这一章将用到OpenCV库来分析图像。OpenCV是世界上最受欢迎的计算机视觉库。由于OpenCV已经为各种不同的平台进行了高度优化，它已然成为业界的事实标准。在继续学习接下来的内容之前，请确保在Python的支持下安装了这个库。你可以在http://opencv.org下载并安装OpenCV。有关各种操作系统的详细安装说明，可以参考网站的文档部分。

9.2 用 OpenCV-Pyhon 操作图像

下面看看如何用OpenCV-Python操作图像。这一节将介绍如何加载并展示图像，并介绍如何裁剪、调整大小以及将图像保存到输出文件中。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import sys

import cv2
import numpy as np
```

(2) 指定输入图像为文件的第一个参数，并使用图像读取函数来读取参数。这个例子中将用到forest.jpg：

```
# 加载并显示图像forest.jpg
input_file = sys.argv[1]
img = cv2.imread(input_file)
```

(3) 显示输入图像：

```
cv2.imshow('Original', img)
```

(4) 现在裁剪该图像。提取输入图像的高度和宽度，然后指定边界：

```
# 裁剪图像
h, w = img.shape[:2]
start_row, end_row = int(0.21*h), int(0.73*h)
start_col, end_col= int(0.37*w), int(0.92*w)
```

(5) 用NumPy式的切分方式裁剪图像，并将其展示出来：

```
img_cropped = img[start_row:end_row, start_col:end_col]
cv2.imshow('Cropped', img_cropped)
```

(6) 将图像大小调整为其原始大小的`1.3`倍，并将其展示出来：

9

```
# 调整图像大小
scaling_factor = 1.3
img_scaled = cv2.resize(img, None, fx=scaling_factor, fy=scaling_ factor,
interpolation=cv2.INTER_LINEAR)
cv2.imshow('Uniform resizing', img_scaled)
```

(7) 之前的方法将均匀地在两个维度上扩展图像。假定我们希望仅在某一个维度进行调整，可以用以下代码实现：

```
img_scaled = cv2.resize(img, (250, 400), interpolation=cv2.INTER_ AREA)
cv2.imshow('Skewed resizing', img_scaled)
```

(8) 将图像保存到输出文件：

```
# 保存图像
output_file = input_file[:-4] + '_cropped.jpg'
cv2.imwrite(output_file, img_cropped)

cv2.waitKey()
```

(9) `waitKey`函数保持显示图像，直到按下键盘上的任一个按键。

(10) 全部代码已经在operating_on_images.py文件中给出。运行该代码，可以看到如图9-1所示的输入图像。



图　9-1

(11) 第二幅输出图像是裁剪后的图像，如图9-2所示。



图 9-2

(12) 第三幅图像是从两个维度均匀地调整大小后的图像，如图9-3所示。



图 9-3

(13) 第四幅图像是仅从一个维度调整大小后的图像，如图9-4所示。



图　9-4

9.3　检测边

边检测是计算机视觉中最常用到的技术之一，常用在很多应用的预处理过程中。接下来介绍如何用不同的边检测器检测输入图像的边。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import sys

import cv2
import numpy as np
```

(2) 加载输入图像，本例用到chair.jpg：

```
# 加载输入图像chair.jpg，转换成灰度图
input_file = sys.argv[1]
img = cv2.imread(input_file, cv2.IMREAD_GRAYSCALE)
```

(3) 提取输入图像的高度和宽度：

```
h, w = img.shape
```

(4) **索贝尔滤波器**（Sobel filter）是一种边检测器，它使用3 × 3内核来检测水平边和垂直边。你可以在http://www.tutorialspoint.com/dip/sobel_operator.htm查看到更多索贝尔滤波器的信息。先运行索贝尔水平检测器：

```
sobel_horizontal = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
```

(5) 运行索贝尔垂直检测器：

```
sobel_vertical = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)
```

(6) **拉普拉斯边检测器**（Laplacian edge detector）可以检测两个方向上的边。你可以在http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm找到更多相关介绍。定义拉普拉斯边检测器如下：

```
laplacian = cv2.Laplacian(img, cv2.CV_64F)
```

(7) 尽管拉普拉斯边检测器弥补了索贝尔边检测器的不足，但是拉普拉斯边检测器的输出仍然带有很多噪声。**Canny边检测器**（Canny edge detector）在解决噪声问题方面优于拉普拉斯边检测器和索贝尔边检测器。Canny边检测器是一个分阶段的处理过程，它用到了迟滞性来做边数据清理。你可以在http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm了解更多相关细节：

```
canny = cv2.Canny(img, 50, 240)
```

(8) 显示所有的输出图像：

```
cv2.imshow('Original', img)
cv2.imshow('Sobel horizontal', sobel_horizontal)
cv2.imshow('Sobel vertical', sobel_vertical)
cv2.imshow('Laplacian', laplacian)
cv2.imshow('Canny', canny)

cv2.waitKey()
```


(9) 全部代码已经在edge_detector.py文件中给出。运行该代码，可以看到原始输入图像如图9-5所示。



图　9-5

(10) 如图9-6所示是索贝尔水平边检测器的输出。注意，它到检测到的边大致都是垂直的，这是因为它是一个水平边检测器，它能检测出在水平方向上的变化。

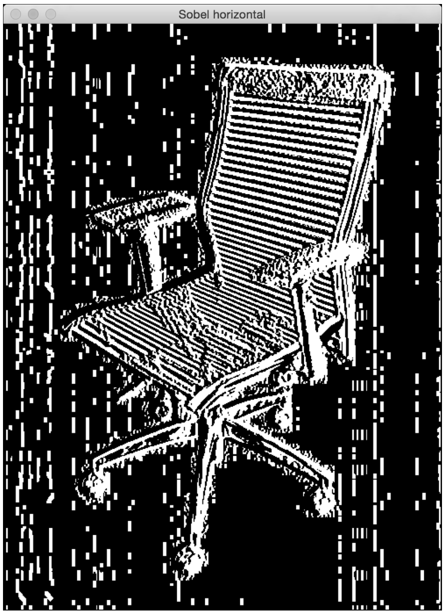


图　9-6

(11) 索贝尔垂直边检测器的输出如图9-7所示。

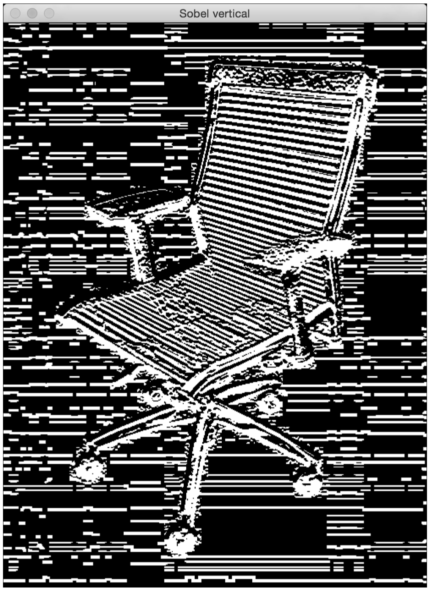


图 9-7

(12) 如图9-8所示是拉普拉斯边检测器的输出。

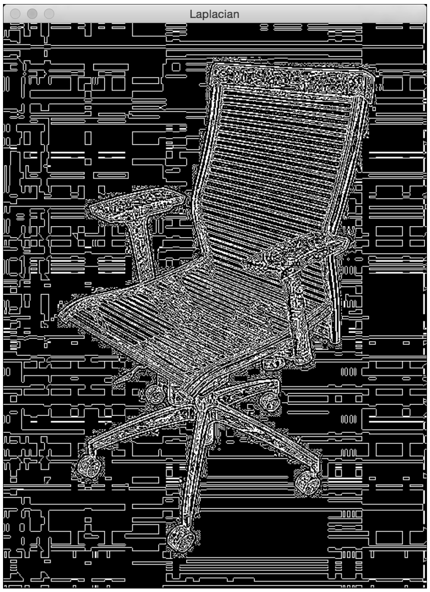


图 9-8

(13) Canny边检测器较好地检测出了所有的边，如图9-9所示。



图　9-9

9.4　直方图均衡化

直方图均衡化是指修改图像的像素以增强图像的对比强度的过程。人的眼睛喜欢对比，这也是为什么几乎所有的照相机系统都会用直方图均衡化来使图像更好看。有趣的是，直方图均衡化过程不同于彩色图像的灰度化过程。在处理彩色图像时有一个问题，在这一节的介绍中将会提到。接下来具体介绍如何实现直方图均衡化。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import sys

import cv2
import numpy as np
```

(2) 加载输入图像。这个例子将用到sunrise.jpg：

```
# 加载输入图像sunrise.jpg
input_file = sys.argv[1]
img = cv2.imread(input_file)
```

(3) 将图像转为灰度并将其显示出来：

```
# 转化为灰度图
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imshow('Input grayscale image', img_gray)
```

(4) 均衡灰度图像的直方图，并将其显示出来：

```
# 均衡直方图
img_gray_histeq = cv2.equalizeHist(img_gray)
cv2.imshow('Histogram equalized - grayscale', img_gray_histeq)
```

(5) 为了均衡彩色图像的直方图，需要用到不同于以上的步骤。直方图均衡化仅适用于亮度通道。一个RGB图像由3个颜色通道组成，因此不能对这些通道单独地做直方图均衡化。在做其他操作之前，需要将强度信息从颜色信息中分离出来。因此，首先将其转换到YUV色彩空间，均衡Y通道，然后将其转换回RGB并得到输出。更多关于YUV色彩空间的详细介绍可查看http://softpixel.com/~cwright/programming/colorspace/yuv。OpenCV默认用BGR格式加载图像，因此需要先将其从BGR转化为YUV：

```
# 均衡彩色图像的直方图
img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
```

(6) 均衡Y通道：

```
img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0])
```

(7) 将其转换回BGR：

```
img_histeq = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
```

(8) 显示输入和输出图像：

```
cv2.imshow('Input color image', img)
cv2.imshow('Histogram equalized - color', img_histeq)

cv2.waitKey()
```

(9) 全部代码在已经histogram_equalizer.py文件中给出，运行该代码，可以看到原始输入图像如图9-10所示。



图　9-10

(10) 直方图均衡化处理后的图像如图9-11所示。



图　9-11

9.5　检测棱角

棱角检测是计算机视觉中的一个重要环节，它帮助我们识别图像中突出的点。这是用于开发图像分析系统中最早期的特征提取技术之一。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import sys

import cv2
import numpy as np
```

(2) 加载输入图像。本例将用到box.png：

```
# 加载图像box.png
input_file = sys.argv[1]
img = cv2.imread(input_file)
cv2.imshow('Input image', img)
```

(3) 将图像转为灰度，并将其强制转换为浮点值。浮点值将用于棱角检测过程：

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_gray = np.float32(img_gray)
```

(4) 对灰度图像运行**哈里斯角检测器**（Harris corner detector）函数。你可以在http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html查看更多哈里斯角检测器的详细介绍：

```
# 哈里斯角检测器
img_harris = cv2.cornerHarris(img_gray, 7, 5, 0.04)
```

(5) 为了标记棱角，需要放大图像：

```
# 放大图像以标记棱角
img_harris = cv2.dilate(img_harris, None)
```

(6) 定义显示重要点个数的阈值：

```
# 用阈值显示棱角
img[img_harris > 0.01 * img_harris.max()] = [0, 0, 0]
```

(7) 显示输出图像：

```
cv2.imshow('Harris Corners', img)
cv2.waitKey()
```

(8) 全部代码已经在corner_detector.py文件中给出。运行该代码，可以看到原始输入图像如图9-12所示。

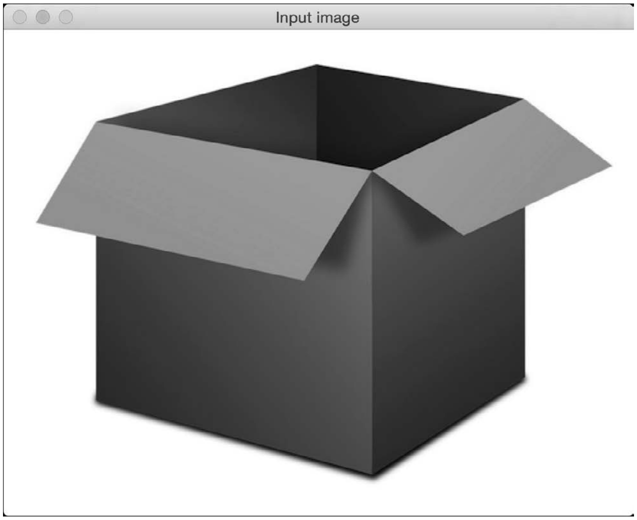


图　9-12

(9) 检测棱角处理后的图像如图9-13所示。

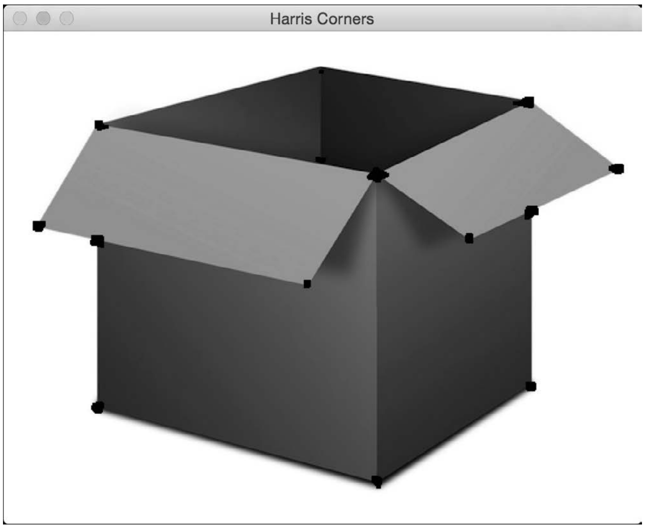


图　9-13

9.6　检测 SIFT 特征点

尺度不变特征变换（Scale Invariant Feature Transform，SIFT）是计算机视觉领域最常用的特征之一。David Lowe首次在其论文中提出该特征，具体可参考https://www.cs.ubc.ca/~lowe/

papers/ijcv04.pdf。此后，SIFT成为图像识别和图像内容分析领域最有效的特征之一，它在大小、方向、对比度等方向都有较强的健壮性。SIFT也是目标识别系统的基础。接下来介绍如何检测这些特征点。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import sys

import cv2
import numpy as np
```

(2) 加载输入图像。本例将用到table.jpg：

```
# 加载图像table.jpg
input_file = sys.argv[1]
img = cv2.imread(input_file)
```

(3) 将图像转为灰度：

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

(4) 初始化SIFT检测器对象并提取关键点：

```
sift = cv2.xfeatures2d.SIFT_create()
keypoints = sift.detect(img_gray, None)
```

(5) 上面所说的关键点是指突出的点，但它们并不是特征。这基本上指出了突出点的位置。SIFT还可以作为非常有效的特征提取器，这一点将在后面的某一节中介绍。

(6) 在输入图像上画出关键点：

```
img_sift = np.copy(img)
cv2.drawKeypoints(img, keypoints, img_sift, flags=cv2.DRAW_
    MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

(7) 显示输入和输出图像：

```
cv2.imshow('Input image', img)
cv2.imshow('SIFT features', img_sift)
cv2.waitKey()
```

(8) 全部代码已经在feature_detector.py文件中给出。运行该代码，可以看到原始输入图像如图9-14所示。



图　9-14

(9) 输出图像如图9-15所示。



图　9-15

9.7　创建 Star 特征检测器

SIFT特征检测器在很多场景中都很好用，但是，当创建目标识别系统时，在用SIFT检测特征之前，可能需要用到一个不同的特征检测器，这使我们能够通过灵活地层叠不同的模块来获得最

佳的性能。这一节将用到**Star特征检测器**（Star feature detector），查看其性能表现。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import sys

import cv2
import numpy as np
```

(2) 定义一个类，用于处理与Star特征检测相关的函数：

```
class StarFeatureDetector(object):
    def __init__(self):
        self.detector = cv2.xfeatures2d.StarDetector_create()
```

(3) 定义一个对输入图像运行检测器的函数：

```
    def detect(self, img):
        return self.detector.detect(img)
```

(4) 在`main`函数中加载输入图像。本例将用到table.jpg：

```
if __name__=='__main__':
    # 加载图像table.jpg
    input_file = sys.argv[1]
    input_img = cv2.imread(input_file)
```

(5) 将图像转为灰度：

```
    # 转为灰度图
    img_gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
```

(6) 用Star特征检测器检测出特征：

```
    # 用Star特征检测器检测出特征
    keypoints = StarFeatureDetector().detect(input_img)
```

(7) 画出输入图像的关键点：

```
    cv2.drawKeypoints(input_img, keypoints, input_img,
               flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

(8) 显示输出图像：

```
    cv2.imshow('Star features', input_img)
    cv2.waitKey()
```

(9) 全部代码已经在star_detector.py文件中给出。运行该代码，可以看到原始输入图像如图9-16所示。



图　9-16

9.8　利用视觉码本和向量量化创建特征

为了创建一个目标识别系统，需要从每张图像中提取特征向量。每张图像需要有一个识别标志，以用于匹配。我们用一个叫**视觉码本**的概念来创建图像识别标志。在训练数据集中，这个码本基本上是一个字典，用于提出关于图像的描述。我们用向量量化方法将很多特征点进行聚类并得出中心点。这些中心点将作为视觉码本的元素，更详细的介绍可以参考http://mi.eng.cam.ac.uk/~cipolla/lectures/PartIB/old/IB-visualcodebook.pdf。

在开始接下来的学习之前，请确保你已经有一些训练图像。本例提供了包含3个类的示例训练数据集，每一类包含20幅图像，这些图像可以在http://www.vision.caltech.edu/html-files/archive.html下载。

为了创建一个健壮的目标识别系统，你需要数万幅图像。该领域有一个非常著名的数据集叫Caltech256，它包括256类图像，每一类都包含上千幅示例图像。你可以在http://www.vision.caltech.edu/Image_Datasets/Caltech256下载该数据集。

详细步骤

(1) 这是一个比较长的例子，因此这里仅介绍一些重点函数。全部代码已经在build_features.py文件中给出。下面先定义一个提取特征的类：

```
class FeatureBuilder(object):
```

(2) 定义一个从输入图像提取特征的方法。下面将用Star检测器获得关键点，然后用SIFT提取这些位置的描述信息：

```
def extract_features(self, img):
    keypoints = StarFeatureDetector().detect(img)
    keypoints, feature_vectors = compute_sift_features(img, keypoints)
    return feature_vectors
```

(3) 从描述信息中提取出中心点：

```
def get_codewords(self, input_map, scaling_size, max_samples=12):
    keypoints_all = []

    count = 0
    cur_class = ''
```

(4) 每幅图像都会生成大量的描述信息。这里将仅用一小部分图像，因为这些中心点并不会发生很大的改变：

```
for item in input_map:
    if count >= max_samples:
        if cur_class != item['object_class']:
            count = 0
    else:
        continue

count += 1
```

(5) 将进程打印出来：

```
    if count == max_samples:
        print "Built centroids for", item['object_class']
```

(6) 提取当前标签：

```
    cur_class = item['object_class']
```

(7) 读取图像并调整其大小：

```
    img = cv2.imread(item['image_path'])
    img = resize_image(img, scaling_size)
```

(8) 设置维度数为128并提取特征：

```
    num_dims = 128
    feature_vectors = self.extract_features(img)
    keypoints_all.extend(feature_vectors)
```

(9) 用向量量化来量化特征点。**向量量化**是一个N维的“四舍五入”，更多详细介绍可参考http://www.data-compression.com/vq.shtml：

```
    kmeans, centroids = BagOfWords().cluster(keypoints_all)
```

9

```
        return kmeans, centroids
```

(10) 定义一个类来处理词袋模型和向量量化：

```
class BagOfWords(object):
    def __init__(self, num_clusters=32):
        self.num_dims = 128
        self.num_clusters = num_clusters
        self.num_retries = 10
```

(11) 定义一个方法来量化数据点。下面将用k-means聚类来实现：

```
    def cluster(self, datapoints):
        kmeans = KMeans(self.num_clusters,
           n_init=max(self.num_retries, 1),
           max_iter=10, tol=1.0)
```

(12) 提取中心点：

```
        res = kmeans.fit(datapoints)
        centroids = res.cluster_centers_
        return kmeans, centroids
```

(13) 定义一个方法来归一化数据：

```
    def normalize(self, input_data):
        sum_input = np.sum(input_data)

        if sum_input > 0:
            return input_data / sum_input
        else:
            return input_data
```

(14) 定义一个方法来获得特征向量：

```
    def construct_feature(self, img, kmeans, centroids):
        keypoints = StarFeatureDetector().detect(img)
        keypoints, feature_vectors = compute_sift_features(img, keypoints)
        labels = kmeans.predict(feature_vectors)
        feature_vector = np.zeros(self.num_clusters)
```

(15) 创建一个直方图并将其归一化：

```
        for i, item in enumerate(feature_vectors):
            feature_vector[labels[i]] += 1

            feature_vector_img = np.reshape(feature_vector,
    ((1, feature_vector.shape[0])))
            return self.normalize(feature_vector_img)
```

(16) 定义一个方法来提取SIFT特征：

```
# 提取SIFT特征
def compute_sift_features(img, keypoints):
```

```
        if img is None:
            raise TypeError('Invalid input image')

        img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        keypoints, descriptors = cv2.xfeatures2d.SIFT_create().compute(img_gray,
keypoints)
        return keypoints, descriptors
```

正如之前提到的，全部代码可参考build_features.py文件。可以用以下方式来运行代码：

```
$ python build_features.py --data-folder /path/to/training_images/ --codebook-file
codebook.pkl --feature-map-file feature_map.pkl
```

结果将产生两个文件，分别为codebook.pkl和feature_map.pkl。下一节中将用到这两个文件。

9.9　用极端随机森林训练图像分类器

本节将用**极端随机森林**（Extremely Random Forests，ERF）来训练图像分类器。一个目标识别系统就是利用图像分类器将图像分到已知的类别中。ERF在机器学习领域非常流行，因为ERF具有较快的速度和比较精确的准确度。我们基于图像的特征构建一组决策树，并通过训练这个森林实现正确决策。更多随机森林的详细内容可参考https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm，更多ERF的内容可参考http://www.montefiore.ulg.ac.be/~ernst/uploads/news/id63/extremely-randomized-trees.pdf。

详细步骤

(1) 创建一个Python文件，并且导入以下程序包：

```
import argparse
import cPickle as pickle

import numpy as np
from sklearn.ensemble import ExtraTreesClassifier
from sklearn import preprocessing
```

(2) 定义一个参数解析器：

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the classifier')
    parser.add_argument("--feature-map-file", dest="feature_map_file",
required=True,
help="Input pickle file containing the feature map")
    parser.add_argument("--model-file", dest="model_file", required=False,
help="Output file where the trained model will be stored")
    return parser
```

(3) 定义一个类来处理ERF训练。这里将用到一个标签编码器来对训练标签进行编码：

```
class ERFTrainer(object):
    def __init__(self, X, label_words):
        self.le = preprocessing.LabelEncoder()
        self.clf = ExtraTreesClassifier(n_estimators=100, max_depth=16,
random_state=0)
```

(4) 对标签编码并训练分类器：

```
        y = self.encode_labels(label_words)
        self.clf.fit(np.asarray(X), y)
```

(5) 定义一个函数，用于对标签进行编码：

```
    def encode_labels(self, label_words):
        self.le.fit(label_words)
        return np.array(self.le.transform(label_words), dtype=np.float32)
```

(6) 定义一个函数，用于将未知数据点进行分类：

```
    def classify(self, X):
        label_nums = self.clf.predict(np.asarray(X))
        label_words = self.le.inverse_transform([int(x) for x in label_nums])
        return label_words
```

(7) 定义`main`函数并解析输入参数：

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    feature_map_file = args.feature_map_file
    model_file = args.model_file
```

(8) 加载上一节中生成的特征地图：

```
    # 加载特征地图
    with open(feature_map_file, 'r') as f:
        feature_map = pickle.load(f)
```

(9) 提取特征向量：

```
    # 提取特征向量和标记
    label_words = [x['object_class'] for x in feature_map]
    dim_size = feature_map[0]['feature_vector'].shape[1]
    X = [np.reshape(x['feature_vector'], (dim_size,)) for x in feature_map]
```

(10) 基于训练数据训练ERF：

```
    # 训练ERF分类器
    erf = ERFTrainer(X, label_words)
```

(11) 保存训练的ERF模型：

```
    if args.model_file:
        with open(args.model_file, 'w') as f:
            pickle.dump(erf, f)
```

(12) 全部代码已经在trainer.py文件中给出。可以用以下方式运行代码：

```
$ python trainer.py --feature-map-file feature_map.pkl --model- file erf.pkl
```

结果将产生一个erf.pkl文件。下一节中将用到该文件。

9.10 创建一个对象识别器

训练好一个ERF模型后，接下来创建一个目标识别器，该识别器可以识别未知图像的内容。

详细步骤

(1) 创建一个Python文件，并且导入以下程序包：

```
import argparse
import cPickle as pickle

import cv2
import numpy as np

import build_features as bf
from trainer import ERFTrainer
```

(2) 定义一个参数解析器：

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Extracts features \
from each line and classifies the data')
    parser.add_argument("--input-image", dest="input_image", required=True,
help="Input image to be classified")
    parser.add_argument("--model-file", dest="model_file", required=True,
help="Input file containing the trained model")
    parser.add_argument("--codebook-file", dest="codebook_file",
required=True, help="Input file containing the codebook")
    return parser
```

(3) 定义一个类来处理图像标签提取函数：

```
class ImageTagExtractor(object):
    def __init__(self, model_file, codebook_file):
        with open(model_file, 'r') as f:
            self.erf = pickle.load(f)

        with open(codebook_file, 'r') as f:
            self.kmeans, self.centroids = pickle.load(f)
```

(4) 定义一个函数，用于使用训练好的ERF模型来预测输出：

```
    def predict(self, img, scaling_size):
```

```
        img = bf.resize_image(img, scaling_size)
        feature_vector = bf.BagOfWords().construct_feature(
img, self.kmeans, self.centroids)
        image_tag = self.erf.classify(feature_vector)[0]
        return image_tag
```

(5) 定义main函数，加载输入图像：

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    model_file = args.model_file
    codebook_file = args.codebook_file
    input_image = cv2.imread(args.input_image)
```

(6) 合理地调整图像大小：

```
    scaling_size = 200
```

(7) 在命令行打印输出结果：

```
    print "\nOutput:", ImageTagExtractor(model_file,
    codebook_file).predict(input_image, scaling_size)
```

(8) 全部代码已经在object_recognizer.py文件中给出。可以用以下方式运行代码：

```
$ python object_recognizer.py --input-image imagefile.jpg --model- file erf.pkl
--codebook-file codebook.pkl
```

可以看到命令行工具中输出的类。

第 10 章

人脸识别

在这一章，我们将介绍以下主题：

- 从网络摄像头采集和处理视频信息
- 用Haar级联创建一个人脸识别器
- 创建一个眼睛和鼻子检测器
- 做主成分分析
- 做核主成分分析
- 做盲源分离
- 用局部二值模式直方图创建一个人脸识别器

10.1 简介

人脸识别是指在给定图像中识别某个人的工作，它不同于从给定图像中定位人脸位置的人脸检测。在人脸检测中，我们不关心这个人是谁，只需要识别包含脸部的图像区域。因此，在一个典型的生物人脸识别系统中，需要在识别脸部之前确定脸部的位置。

人脸识别对人类来说非常容易，毫不费力就可以做到，并且我们一直都在这样做，但是如何才能让机器做同样的事情呢？我们需要了解会利用脸部的哪个部分来识别一个人。人类的大脑有一个内部结构，它似乎可以对特定的特征做出相应的反应，例如边、角度、情绪等。人类的视觉皮层将这些特征综合起来，做出一个连贯性推断。如果希望机器也能同样精确地识别人脸，那就需要用同样的方式模拟这个问题。下面需要从输入的图像中提取相关特征，并把它转换成一个有意义的表示形式。

10.2 从网络摄像头采集和处理视频信息

本节将用网络摄像头采集视频数据。下面来看如何用OpenCV-Python从网络摄像头采集视频信息。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import cv2
```

(2) OpenCV提供了一个视频采集对象，可以利用该对象从网络摄像头采集图像。0输入参数指定网络摄像头的ID。如果连接的是USB摄像头，将有一个不同的ID：

```
# 初始化网络摄像头
cap = cv2.VideoCapture(0)
```

(3) 定义网络摄像头采集的帧的比例系数：

```
# 定义网络摄像头采集图像的比例系数
scaling_factor = 0.5
```

(4) 启动一个无限循环来采集帧，直到按下Esc键。从网络摄像头读取帧：

```
# 循环采集直到按下Esc键
while True:
    # 采集当前画面
    ret, frame = cap.read()
```

(5) 调整帧的大小不是必须的，但是这在编写代码中很重要：

```
    # 调整帧的大小
    frame = cv2.resize(frame, None, fx=scaling_factor, fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

(6) 显示帧：

```
    # 显示帧
    cv2.imshow('Webcam', frame)
```

(7) 等待1 ms，然后采集下一帧：

```
    # 检查是否按了Esc键
    c = cv2.waitKey(1)
    if c == 27:
        break
```

(8) 释放视频采集对象：

```
# 释放视频采集对象
cap.release()
```

(9) 在结束代码之前关闭所有活动窗体：

```
# 关闭所有活动窗体
cv2.destroyAllWindows()
```

(10) 全部代码已经包含在video_capture.py文件中。运行该代码，可以从网络摄像头中看到类似如图10-1所示的图像。



图 10-1

10.3 用 Haar 级联创建一个人脸识别器

正如前面讨论过的，人脸检测是确定输入图像中人脸位置的过程。我们将用**Haar级联**来做人脸检测。Haar级联通过在多个尺度上从图像中提取大量的简单特征来实现。简单特征主要指边、线、矩形特征等，这些特征都非常易于计算，然后通过创建一系列简单的分类器来做训练。使用**自适应增强**技术可以使得这个过程更健壮，更多细节可以查看http://docs.opencv.org/3.1.0/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0。下面来看如何在网络摄像头采集的视频帧中确定人脸位置。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import cv2
import numpy as np
```

(2) 加载人脸检测级联文件。这是可以用作检测器的训练模型：

```
# 导入人脸检测级联文件
face_cascade = cv2.CascadeClassifier('cascade_files/haarcascade_
frontalface_alt.xml')
```

(3) 确定级联文件是否正确地加载：

```
# 确定级联文件是否正确地加载
if face_cascade.empty():
    raise IOError('Unable to load the face cascade classifier xml file')
```

(4) 生成一个视频采集对象：

```
# 初始化视频采集对象
cap = cv2.VideoCapture(0)
```

(5) 定义图像向下采样的比例系数：

```
# 定义图像向下采样的比例系数
scaling_factor = 0.5
```

(6) 循环采集直到按下Esc键：

```
# 循环采集直到按下Esc键
while True:
    # 采集当前帧并进行调整
    ret, frame = cap.read()
```

(7) 调整帧的大小：

```
    frame = cv2.resize(frame, None, fx=scaling_factor, fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

(8) 将图像转为灰度图。这里需要灰度图像来运行人脸检测器：

```
    # 将图像转为灰度图
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

(9) 在灰度图像上运行人脸检测器。参数1.3是指每个阶段的乘积系数。参数5是指每个候选矩形应该拥有的最小近邻数量，这样我们可以维持这一数量。候选矩形是指人脸可能被检测到的候选区域：

```
    # 在灰度图像上运行人脸检测器
    face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
```

(10) 对于每个检测到的人脸区域，在其周围画出矩形：

```
    # 在脸部画出矩形
    for (x,y,w,h) in face_rects:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
```

(11) 展示输出图像：

```
    # 展示输出图像
    cv2.imshow('Face Detector', frame)
```

(12) 在下一次迭代之前等待1 ms，如果用户按下Esc键，就跳出循环：

```
    # 检查是否按下Esc键
    c = cv2.waitKey(1)
```

```
    if c == 27:
        break
```

(13) 在结束代码之前，释放并销毁所有对象：

```
# 释放视频采样对象并关闭窗口
cap.release()
cv2.destroyAllWindows()
```

(14) 全部代码已经包含在face_detector.py文件中。运行该代码，可以看到从网络摄像视频文件中人脸被检测出来了，如图10-2所示。

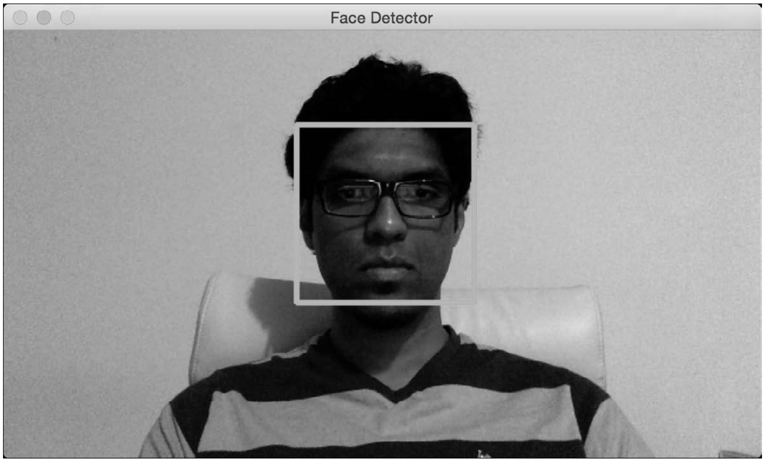


图　10-2

10.4　创建一个眼睛和鼻子检测器

Haar级联方法可以被扩展应用于各种对象的检测。接下来看看如何利用该方法检测输入视频文件中的眼睛和鼻子。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import cv2
import numpy as np
```

(2) 加载人脸、眼睛和鼻子级联文件：

```
# 加载人脸、眼睛和鼻子级联文件
face_cascade = cv2.CascadeClassifier('cascade_files/haarcascade_
```

```
frontalface_alt.xml')
eye_cascade = cv2.CascadeClassifier('cascade_files/haarcascade_eye.xml')
nose_cascade = cv2.CascadeClassifier('cascade_files/haarcascade_mcs_nose.xml')
```

(3) 确定级联文件是否正确地加载：

```
# 检查脸部级联文件是否加载
if face_cascade.empty():
    raise IOError('Unable to load the face cascade classifier xml file')

# 检查眼睛级联文件是否加载
if eye_cascade.empty():
    raise IOError('Unable to load the eye cascade classifier xml file')

# 检查鼻子级联文件是否加载
if nose_cascade.empty():
    raise IOError('Unable to load the nose cascade classifier xml file')
```

(4) 初始化视频采集对象：

```
# 初始化视频采集对象并定义比例系数
cap = cv2.VideoCapture(0)
```

(5) 定义比例系数：

```
scaling_factor = 0.5
```

(6) 重复循环直至用户按下Esc键：

```
while True:
    # 读取当前帧画面，调整大小，转为灰度图
    ret, frame = cap.read()
```

(7) 调整帧的大小：

```
    frame = cv2.resize(frame, None, fx=scaling_factor, fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

(8) 将图像转为灰度图：

```
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

(9) 在灰度图像上运行人脸检测器：

```
    # 在灰度图像上运行人脸检测器
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

(10) 因为眼睛和鼻子总是位于脸部区域，所以这里仅在脸部区域运行检测器：

```
    # 在每张脸的矩形区域运行眼睛和鼻子检测器
    for (x,y,w,h) in faces:
```

(11) 提取人脸ROI信息：

```
        # 从彩色与灰度图中提取人脸ROI信息
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = frame[y:y+h, x:x+w]
```

(12) 运行眼睛检测器：

```
        # 在灰度图ROI信息中检测眼睛
        eye_rects = eye_cascade.detectMultiScale(roi_gray)
```

(13) 运行鼻子检测器：

```
        # 在灰度图ROI信息中检测鼻子
        nose_rects = nose_cascade.detectMultiScale(roi_gray, 1.3, 5)
```

(14) 在眼睛周围画圈：

```
       # 在眼睛周围画绿色的圈
        for (x_eye, y_eye, w_eye, h_eye) in eye_rects:
            center = (int(x_eye + 0.5*w_eye), int(y_eye + 0.5*h_eye))
            radius = int(0.3 * (w_eye + h_eye))
            color = (0, 255, 0)
            thickness = 3
            cv2.circle(roi_color, center, radius, color, thickness)
```

(15) 在鼻子周围画矩形：

```
        for (x_nose, y_nose, w_nose, h_nose) in nose_rects:
            cv2.rectangle(roi_color, (x_nose, y_nose), (x_nose+w_nose,
                y_nose+h_nose), (0,255,0), 3)
            break
```

(16) 展示该图像：

```
    # 展示图像
    cv2.imshow('Eye and nose detector', frame)
```

(17) 在下一次迭代之前等待1 ms，如果用户按下Esc键，就跳出循环：

```
    # 检查是否按了Esc键
    c = cv2.waitKey(1)
    if c == 27:
        break
```

(18) 在结束代码之前，释放并销毁所有对象：

```
# 释放视频采样对象并关闭窗口
cap.release()
cv2.destroyAllWindows()
```

(19) 全部代码已经包含在eye_nose_detector.py文件中。运行该代码，可以看到网络摄像视频文件中的眼睛和鼻子被检测出来了，如图10-3所示。

10



图 10-3

10.5 做主成分分析

主成分分析（Principal Components Analysis，PCA）是一个降低维度的技术，常用于计算机视觉和机器学习中。当需要处理很大的特征维度时，训练机器学习系统变得异常昂贵，因此需要在训练系统之前降低数据的维度。但是，降低维度时，我们并不想损失数据中的重要信息，此时PCA便是最佳选择。PCA识别数据中的重要成分，并将其按照重要程度排序。你可以在http://dai.fmph.uniba.sk/courses/ml/sl/PCA.pdf中了解更多细节。PCA也常用于人脸识别系统。接下来看看如何对输入数据做主成分分析。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
from sklearn import decomposition
```

(2) 为输入数据定义5个维度。前两个维度是独立的，但是后3个维度将依赖于前两个维度。也就是说，可以去掉后3个维度，因为它们并没有提供任何新信息：

```
# 定义特征
x1 = np.random.normal(size=250)
x2 = np.random.normal(size=250)
x3 = 2*x1 + 3*x2
x4 = 4*x1 - x2
x5 = x3 + 2*x4
```

(3) 创建一个带这些特征的数据集：

```
# 创建特征数据集
X = np.c_[x1, x3, x2, x5, x4]
```

(4) 创建一个PCA对象：

```
# 创建一个PCA
pca = decomposition.PCA()
```

(5) 对输入数据做主成分分析（PCA）：

```
pca.fit(X)
```

(6) 打印维度的方差：

```
# 打印方差
variances = pca.explained_variance_
print '\nVariances in decreasing order:\n', variances
```

(7) 如果一个特定的维度是有用的，那么它应该有一个有意义的方差值。设置一个阈值并确定重要的维度：

```
# 找出有用的维度数量
thresh_variance = 0.8
num_useful_dims = len(np.where(variances > thresh_variance)[0])
print '\nNumber of useful dimensions:', num_useful_dims
```

(8) 正如之前提到的，PCA识别只有两个维度在这个数据集中很重要：

```
# 只有两个维度是有效的
pca.n_components = num_useful_dims
```

(9) 将数据集从5维转换为二维：

```
X_new = pca.fit_transform(X)
print '\nShape before:', X.shape
print 'Shape after:', X_new.shape
```

(10) 全部代码已经包含在pca.py文件中。运行该代码，可以在终端看到如图10-4所示的结果。

```
Variances in decreasing order:
[  1.13489352e+02   1.08125265e+01   3.34017371e-31   4.36320756e-32
   1.49223239e-32]

Number of useful dimensions: 2

Shape before: (250, 5)
Shape after: (250, 2)
```

图　10-4

10.6　做核主成分分析

PCA能很好地降低维度，但PCA是以线性方式工作的，如果数据集不是以线性方式组织的，

那么PCA并不能实现其功能。但是核主成分分析可以很好地解决这个问题，关于核主成分分析的细节介绍可参考http://www.ics.uci.edu/~welling/classnotes/papers_class/Kernel-PCA.pdf。接下来看看如何对输入数据做核主成分分析，同时将其与主成分分析进行对比。

详细步骤

(1) 创建一个Python文件，并且导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles
```

(2) 定义随机数发生器的种子值，以便产生用于分析的数据示例：

```
# 定义随机数发生器的种子值
np.random.seed(7)
```

(3) 生成以同心圆分布的数据，以演示PCA在这种情况下是如何工作的：

```
# 生成样本
X, y = make_circles(n_samples=500, factor=0.2, noise=0.04)
```

(4) 对这组数据做主成分分析：

```
# 做主成分分析
pca = PCA()
X_pca = pca.fit_transform(X)
```

(5) 对输入数据做核主成分分析：

```
# 做核主成分分析
kernel_pca = KernelPCA(kernel="rbf", fit_inverse_transform=True, gamma=10)
X_kernel_pca = kernel_pca.fit_transform(X)
X_inverse = kernel_pca.inverse_transform(X_kernel_pca)
```

(6) 画出原始输入数据：

```
# 画出原始输入数据
class_0 = np.where(y == 0)
class_1 = np.where(y == 1)
plt.figure()
plt.title("Original data")
plt.plot(X[class_0, 0], X[class_0, 1], "ko", mfc='none')
plt.plot(X[class_1, 0], X[class_1, 1], "kx")
plt.xlabel("1st dimension")
plt.ylabel("2nd dimension")
```

(7) 画出主成分分析后的数据：

```
# 画出主成分分析后的数据
plt.figure()
```

```
plt.plot(X_pca[class_0, 0], X_pca[class_0, 1], "ko", mfc='none')
plt.plot(X_pca[class_1, 0], X_pca[class_1, 1], "kx")
plt.title("Data transformed using PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd principal component")
```

(8) 画出核主成分分析后的数据：

```
# 画出核主成分分析后的数据
plt.figure()
plt.plot(X_kernel_pca[class_0, 0], X_kernel_pca[class_0, 1], "ko", mfc='none')
plt.plot(X_kernel_pca[class_1, 0], X_kernel_pca[class_1, 1], "kx")
plt.title("Data transformed using Kernel PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd principal component")
```

(9) 用核方法将数据转换回原始空间，查看是否存在这样的可逆关系：

```
# 用核方法将数据转换回原始空间
plt.figure()
plt.plot(X_inverse[class_0, 0], X_inverse[class_0, 1], "ko", mfc='none')
plt.plot(X_inverse[class_1, 0], X_inverse[class_1, 1], "kx")
plt.title("Inverse transform")
plt.xlabel("1st dimension")
plt.ylabel("2nd dimension")

plt.show()
```

(10) 全部代码已经包含在kpca.py文件中。运行该代码，可以看到4幅图像。第一幅图像是原始数据，如图10-5所示。

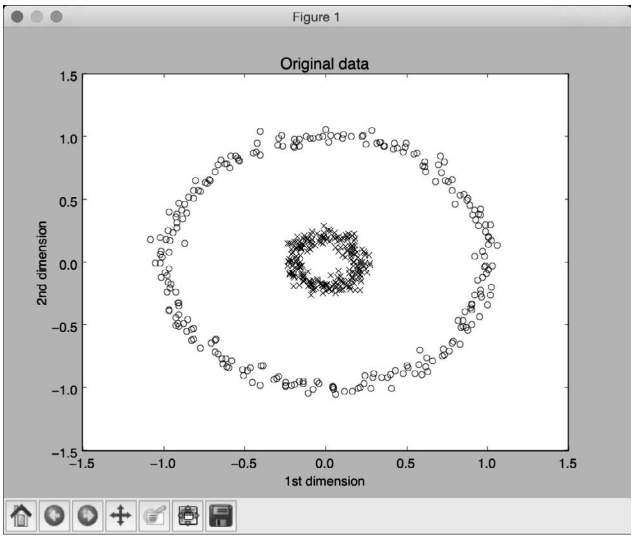


图 10-5

第二幅图像是运行主成分分析后的数据，如图10-6所示。

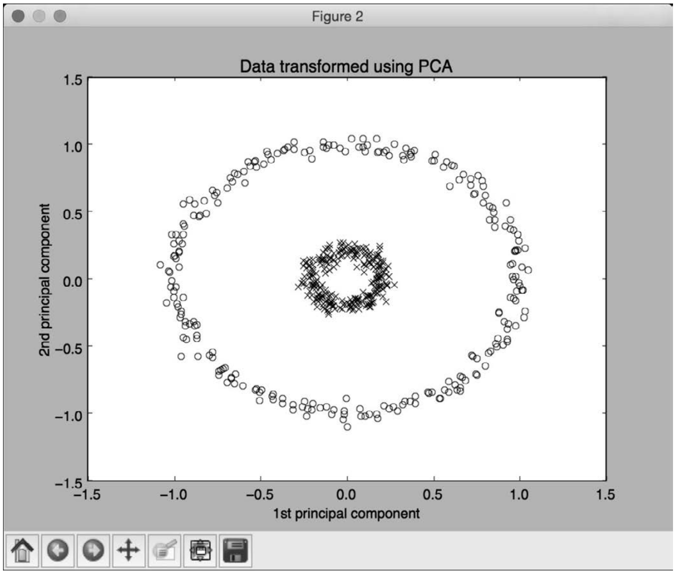


图　10-6

第三幅图像是运行核主成分分析后的数据，如图10-7所示。注意，点都聚集到图像的右半部分。

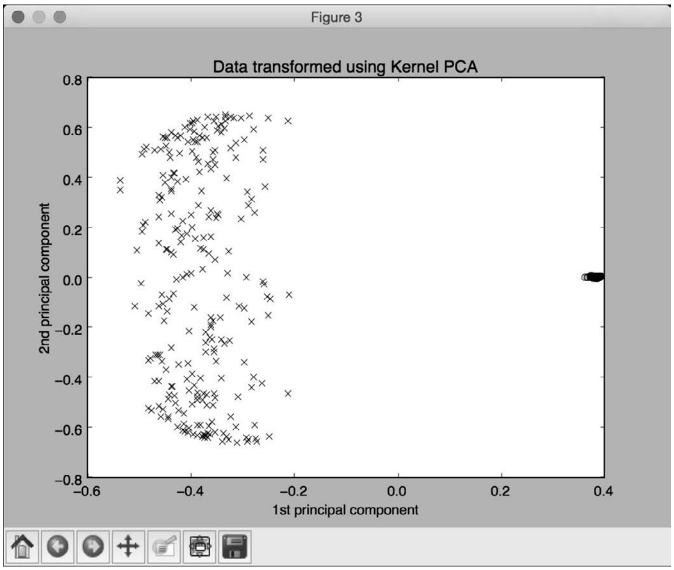


图　10-7

第四幅图像展示了将数据逆变换回其原始空间，如图10-8所示。

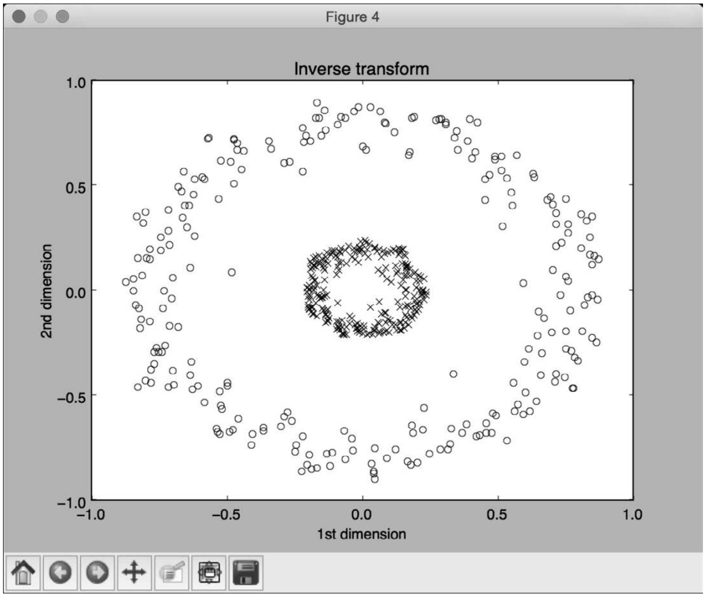


图　10-8

10.7　做盲源分离

盲源分离是指将信号从混合体中分离出来的过程。假设一组不同的信号发生器生成了不同的信号，而一个公共接收机接收到了所有这些信号。现在，我们的工作是利用这些信号的性质将这些信号从混合体中分离出来。我们将用**独立成分分析**（Independent Components Analysis，ICA）算法来实现。关于独立成分分析的细节可查看http://www.mit.edu/~gari/teaching/6.555/LECTURE_NOTES/ch15_bss.pdf。接下来看看如何实现。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

from sklearn.decomposition import PCA, FastICA
```

(2) 本例将用到mixture_of_signals.txt文件提供的数据。加载该数据：

```
# 加载数据
input_file = 'mixture_of_signals.txt'
X = np.loadtxt(input_file)
```

(3) 创建ICA对象：

```
# 计算ICA
ica = FastICA(n_components=4)
```

(4) 基于ICA重构信号：

```
# 重构信号
signals_ica = ica.fit_transform(X)
```

(5) 提取混合矩阵：

```
# 提取混合矩阵
mixing_mat = ica.mixing_
```

(6) 执行PCA做对比：

```
# 执行PCA
pca = PCA(n_components=4)
signals_pca = pca.fit_transform(X)  # 基于正交成分重构信号
```

(7) 定义信号列表来将其画出：

```
# 定义画图参数
models = [X, signals_ica, signals_pca]
```

(8) 指定颜色：

```
colors = ['blue', 'red', 'black', 'green']
```

(9) 画出输入信号：

```
# 画出输入信号
plt.figure()
plt.title('Input signal (mixture)')
for i, (sig, color) in enumerate(zip(X.T, colors), 1):
    plt.plot(sig, color=color)
```

(10) 画出利用ICA分离的信号：

```
# 画出利用ICA分离的信号
plt.figure()
plt.title('ICA separated signals')
plt.subplots_adjust(left=0.1, bottom=0.05, right=0.94,
        top=0.94, wspace=0.25, hspace=0.45)
```

(11) 用不同的颜色画出子图：

```
for i, (sig, color) in enumerate(zip(signals_ica.T, colors), 1):
    plt.subplot(4, 1, i)
```

```
    plt.title('Signal ' + str(i))
    plt.plot(sig, color=color)
```

(12) 画出用PCA分离的信号：

```
# 画出PCA信号
plt.figure()
plt.title('PCA separated signals')
plt.subplots_adjust(left=0.1, bottom=0.05, right=0.94,
        top=0.94, wspace=0.25, hspace=0.45)
```

(13) 用不同的颜色画出各子图：

```
for i, (sig, color) in enumerate(zip(signals_pca.T, colors), 1):
    plt.subplot(4, 1, i)
    plt.title('Signal ' + str(i))
    plt.plot(sig, color=color)

plt.show()
```

(14) 全部代码已经包含在blind_source_separation.py文件中。运行该代码，可以看到3幅图像。第一幅图像是原始数据，也就是信号的混合体，如图10-9所示。

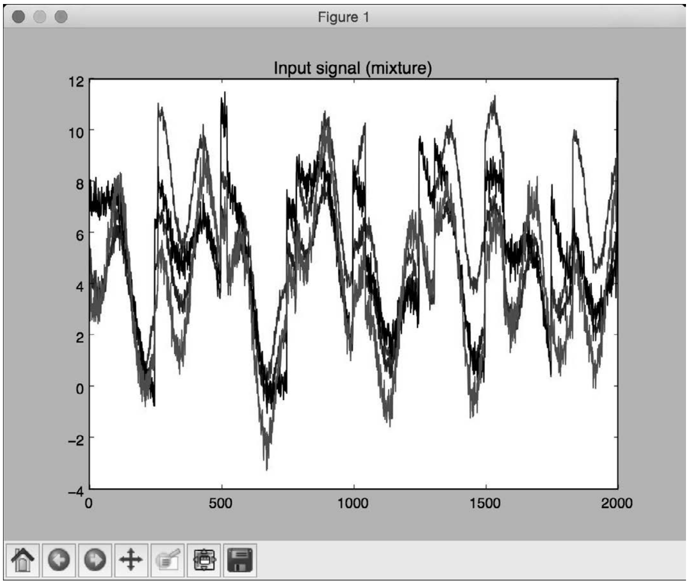


图　10-9

第二幅图像是用ICA分离的信号，如图10-10所示。

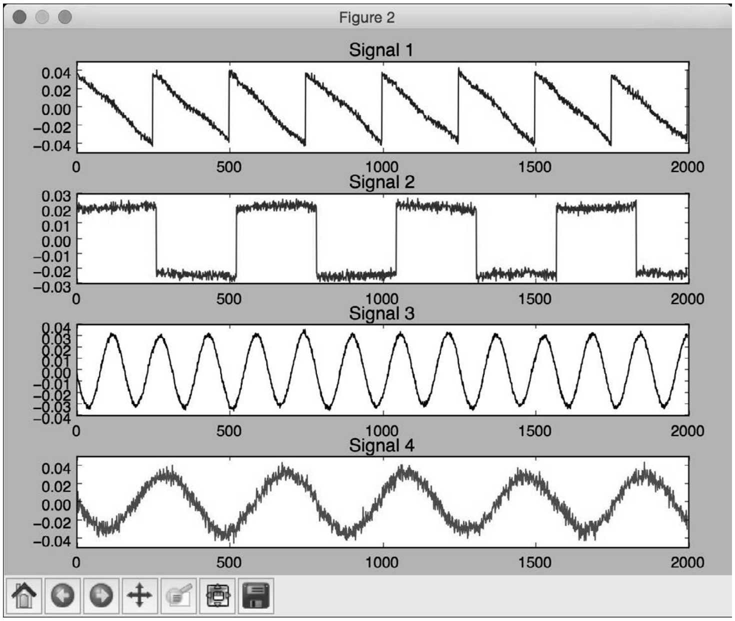


图　10-10

第三幅图像是用PCA分离的信号，如图10-11所示。

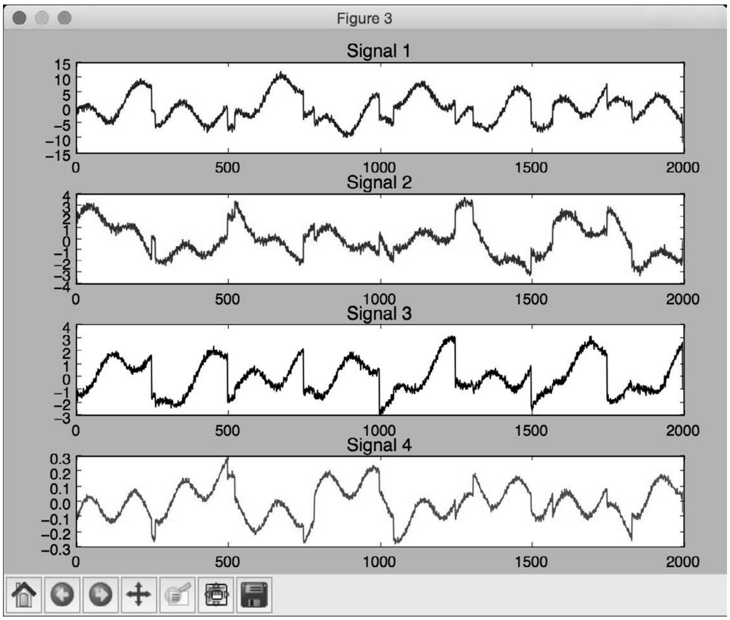


图　10-11

10.8　用局部二值模式直方图创建一个人脸识别器

现在已经准备好创建人脸识别器了。接下来需要一个人脸数据集来做训练，所以这里提供了一个faces_dataset文件夹，该文件夹中包含足够的图像可以用来做训练。该数据集是http://www.vision.caltech.edu/Image_Datasets/faces/faces.tar给出的一个子集，包含了一定数量的图像，可以利用这些图像来训练一个人脸识别系统。

我们将用**局部二值模式直方图**（Local Binary Patterns Histograms）创建人脸识别系统。在数据集中，你可以看到不同的人。接下来的工作是构建一个能将每个人从其他人中区分出来的系统。如果看到从未见过的图像，系统会将其分派到已有的类中。更多关于局部二值模式直方图的细节信息可查看http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html#local-binary-patterns-histograms。下面看看如何创建一个人脸识别器。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import os

import cv2
import numpy as np
from sklearn import preprocessing
```

(2) 定义一个类来处理与类标签编码相关的所有任务：

```
# 定义一个类来处理与类标签编码相关的所有任务
class LabelEncoder(object):
```

(3) 定义一个方法来为这些标签编码。在输入训练数据中，标签用单词表示，但我们需要数字来训练系统。该方法将定义一个预处理对象，该对象将单词转换成数字，同时保留这种前向后向的映射关系：

```
    # 将单词转换成数字的编码方法
    def encode_labels(self, label_words):
        self.le = preprocessing.LabelEncoder()
        self.le.fit(label_words)
```

(4) 定义一个将单词转换成数字的方法：

```
    # 将输入单词转换成数字
    def word_to_num(self, label_word):
        return int(self.le.transform([label_word])[0])
```

(5) 定义一个方法，用于将数字转换回其原始单词：

```
    # 将数字转换为单词
```

```
    def num_to_word(self, label_num):
        return self.le.inverse_transform([label_num])[0]
```

(6) 定义一个方法，用于从输入文件夹中提取图像和标签：

```
# 从输入文件夹中提取图像和标签
def get_images_and_labels(input_path):
    label_words = []
```

(7) 对输入文件夹做递归迭代，提取所有图像的路径：

```
    # 对输入文件夹做递归迭代并追加文件
    for root, dirs, files in os.walk(input_path):
        for filename in (x for x in files if x.endswith('.jpg')):
            filepath = os.path.join(root, filename)
            label_words.append(filepath.split('/')[-2])
```

(8) 初始化变量：

```
    # 初始化变量
    images = []
    le = LabelEncoder()
    le.encode_labels(label_words)
    labels = []
```

(9) 为训练解析输入目录：

```
    # 解析输入目录
    for root, dirs, files in os.walk(input_path):
        for filename in (x for x in files if x.endswith('.jpg')):
            filepath = os.path.join(root, filename)
```

(10) 将当前图像读取成灰度格式：

```
    # 将当前图像读取成灰度格式
    image = cv2.imread(filepath, 0)
```

(11) 从文件夹路径中提取标签：

```
    # 从文件夹路径中提取标签
    name = filepath.split('/')[-2]
```

(12) 对该图像做人脸检测：

```
    # 做人脸检测
    faces = faceCascade.detectMultiScale(image, 1.1, 2, minSize=(100,100))
```

(13) 提取ROI属性值，并将这些值和标签编码器返回：

```
            # 循环处理每一张脸
            for (x, y, w, h) in faces:
                images.append(image[y:y+h, x:x+w])
                labels.append(le.word_to_num(name))

      return images, labels, le
```

(14) 定义main函数，并定义人脸级联文件的路径：

```
if __name__=='__main__':
    cascade_path = "cascade_files/haarcascade_frontalface_alt.xml"
    path_train = 'faces_dataset/train'
    path_test = 'faces_dataset/test'
```

(15) 加载人脸级联文件：

```
    # 加载人脸级联文件
    faceCascade = cv2.CascadeClassifier(cascade_path)
```

(16) 生成局部二值模式直方图人脸识别器对象：

```
    # 生成局部二值模式直方图人脸识别器
    recognizer = cv2.face.createLBPHFaceRecognizer()
```

(17) 为输入路径提取图像、标签和标签编码器：

```
    # 从训练数据集中提取图像、标签和标签编码器
    images, labels, le = get_images_and_labels(path_train)
```

(18) 用提取的数据训练人脸识别器：

```
    # 训练人脸识别器
    print "\nTraining..."
    recognizer.train(images, np.array(labels))
```

(19) 用未知数据测试人脸识别器：

```
    # 用未知数据测试人脸识别器
    print '\nPerforming prediction on test images...'
    stop_flag = False
    for root, dirs, files in os.walk(path_test):
        for filename in (x for x in files if x.endswith('.jpg')):
            filepath = os.path.join(root, filename)
```

(20) 加载图像：

```
            # 读取图像
            predict_image = cv2.imread(filepath, 0)
```

(21) 用人脸检测器确定人脸的位置：

```
            # 检测人脸
            faces = faceCascade.detectMultiScale(predict_image, 1.1,
                    2, minSize=(100,100))
```

(22) 对于每个人脸ROI，运行人脸识别器：

```
            # 循环处理每一张脸
            for (x, y, w, h) in faces:
                # Predict the output
```

```
                predicted_index, conf = recognizer.predict(
                        predict_image[y:y+h, x:x+w])
```

(23) 将标签转换为单词：

```
                # 将标签转换为单词
                predicted_person = le.num_to_word(predicted_index)
```

(24) 在输出图像中叠加文字，并将其展示：

```
                # 在输出图像中叠加文字，并显示图像
                cv2.putText(predict_image, 'Prediction: ' + predicted_person,
                        (10,60), cv2.FONT_HERSHEY_SIMPLEX, 2, (255,255,255), 6)
                cv2.imshow("Recognizing face", predict_image)
```

(25) 检查用户是否按下Esc键。如果有，则跳出循环：

```
            c = cv2.waitKey(0)
            if c == 27:
                stop_flag = True
                break

        if stop_flag:
            break
```

(26) 全部代码已经包含在face_recognizer.py文件中。运行该代码，可以得到一个输出窗体，窗体中显示测试图像的预测输出。按下空格键可以继续循环。测试图像中有3个不同的人。第一个人的输出结果如图10-12所示。

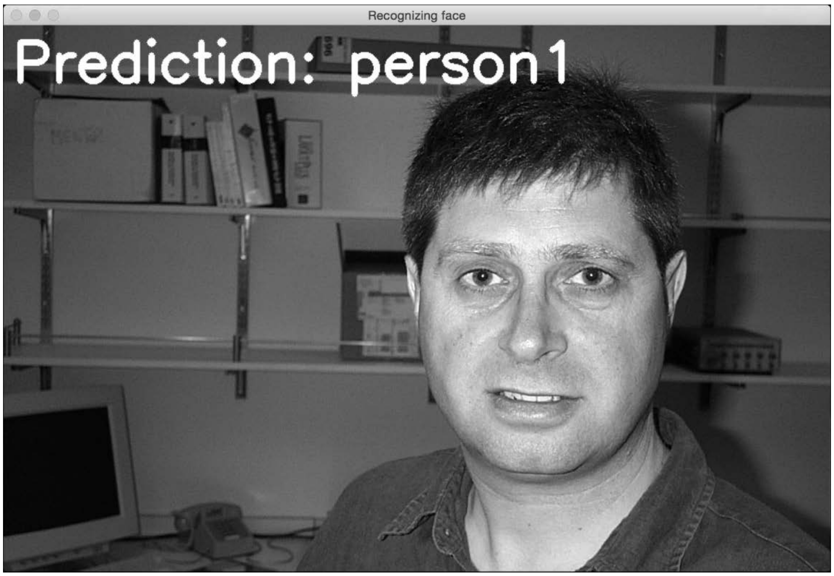


图　10-12

第二个人的输出结果如图10-13所示。

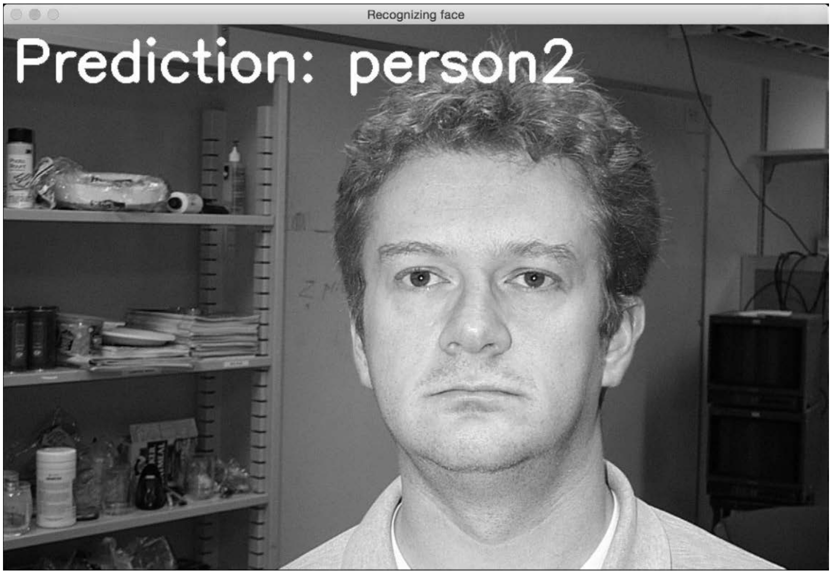


图　10-13

第三个人的输出结果如图10-14所示。

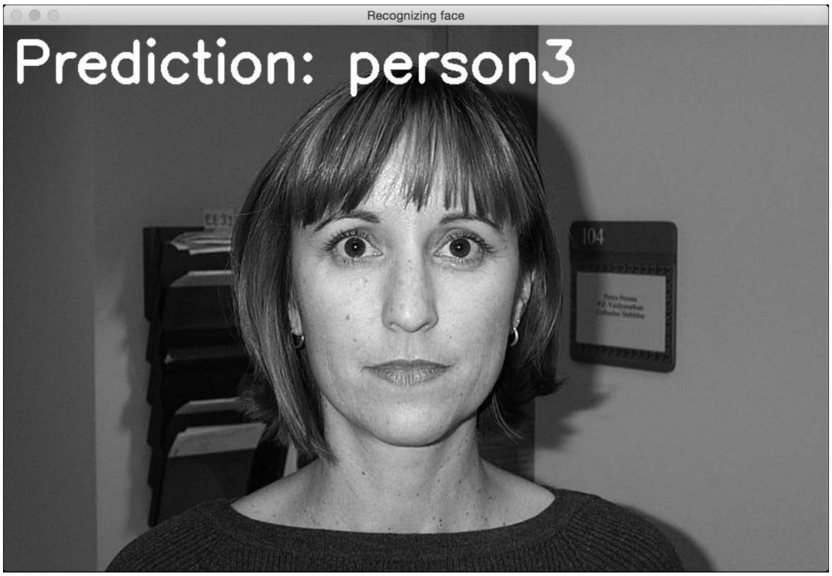


图　10-14

第 11 章

深度神经网络

在这一章，我们将介绍以下主题：

- 创建一个感知器
- 创建一个单层神经网络
- 创建一个深度神经网络
- 创建一个向量量化器
- 为序列数据分析创建一个递归神经网络
- 在光学字符识别数据库中将字符可视化
- 用神经网络创建一个光学字符识别器

11.1 简介

人类的大脑很擅长于鉴别和识别物体，我们希望机器也可以做同样的事情。一个神经网络就是一个模仿人类大脑激发学习过程的框架。神经网络被用于从数据中识别隐藏的模式。正如所有的学习算法，神经网络处理的是数字。因此，如果想要实现处理现实世界中任何包含图像、文字、传感器等的任务，就必须将其转换成数值形式，然后将其输入到一个神经网络。我们可以用神经网络做分类、聚类、生成以及其他相关的任务。

神经网络由一层层**神经元**组成。这些神经元模拟人类大脑中的生物神经元。每一层都是一组独立的神经元，这些神经元与相邻层的神经元相连。输入层对应我们提供的输入数据，而输出层包括了我们期望的输出结果。输入层与输出层之间的层统称为**隐藏层**。如果设计的神经网络包括多个隐藏层，那么就能通过这些层的自我训练获得更大的精确度。

假设我们希望神经网络按照我们的要求来对数据进行分类。为了使神经网络完成相应的任务，需要提供带标签的训练数据。神经网络将通过优化成本函数来训练自己。我们不停地迭代，直到错误率下降到一定的阈值为止。

那么“深度”神经网络是什么？深度神经网络是由多个隐藏层组成的神经网络。一般来说，

这就属于深度学习的范畴。深度学习用于研究这些神经网络，而这些神经网络由多个层次的多层结构组成。

你可以在http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html查看神经网络的教程。本章中将用到NeuroLab库。在接下来的学习之前，请确保你已经安装了这个库，安装指导可以参考https://pythonhosted.org/neurolab/install.html。接下来看看如何设计和开发这些神经网络。

11.2 创建一个感知器

让我们通过感知器开始神经网络之旅。**感知器**是一个单独的神经元，它负责执行所有的计算。这是一个非常简单的模型，但是它奠定了构建复杂神经网络的基础。该模型如图11-1所示。

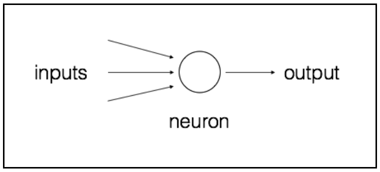


图 11-1

该神经元将多个输入用不同的权重系数融合起来，并加上偏差值来计算输出。这是一个简单的线性方程，它将输入值与感知器的输出关联起来。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import neurolab as nl
import matplotlib.pyplot as plt
```

(2)定义一些输入数据及其对应的标签：

```
# 定义输入数据
data = np.array([[0.3, 0.2], [0.1, 0.4], [0.4, 0.6], [0.9, 0.5]])
labels = np.array([[0], [0], [0], [1]])
```

(3) 将这些点画出，以查看这些点的布局：

```
# 画出输入数据
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Input data')
```

(4) 定义一个感知器`perceptron`，它有两个输入。该函数还需要限定输入数据的最大值和最小值：

```
# 定义有两个输入的感知器，在感知器第一个参数的每个元素中指定参数的最大值和最小值
perceptron = nl.net.newp([[0, 1],[0, 1]], 1)
```

(5) 接下来训练该感知器。epochs的数量指定了训练数据集需要完成的测试次数。`show`参数指定了显示训练过程的频率。`lr`参数指定了感知器的学习速度。学习速度是指学习算法在参数空间中搜索的步长。如果这个值太大，算法行进得会很快，但可能会错失最优值。而如果这个值太小，则该算法可以命中最优值，但是算法行进得会很慢，所以需要进行权衡。这里取`0.01`：

```
# 训练感知器
error = perceptron.train(data, labels, epochs=50, show=15, lr=0.01)
```

(6) 画出结果：

```
# 画出结果
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.grid()
plt.title('Training error progress')

plt.show()
```

(7) 全部代码已经在perceptron.py文件中给出。运行该代码，可以看到两幅图像。第一幅图像显示输入数据，如图11-2所示。

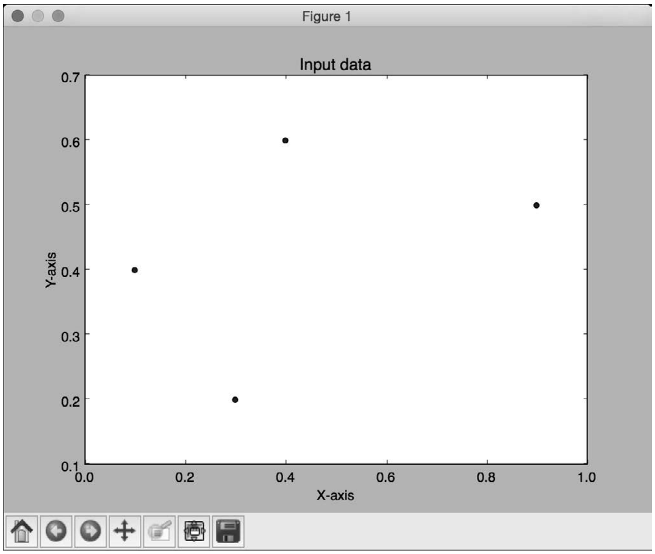


图　11-2

第二幅图像显示训练误差进程，如图11-3所示。

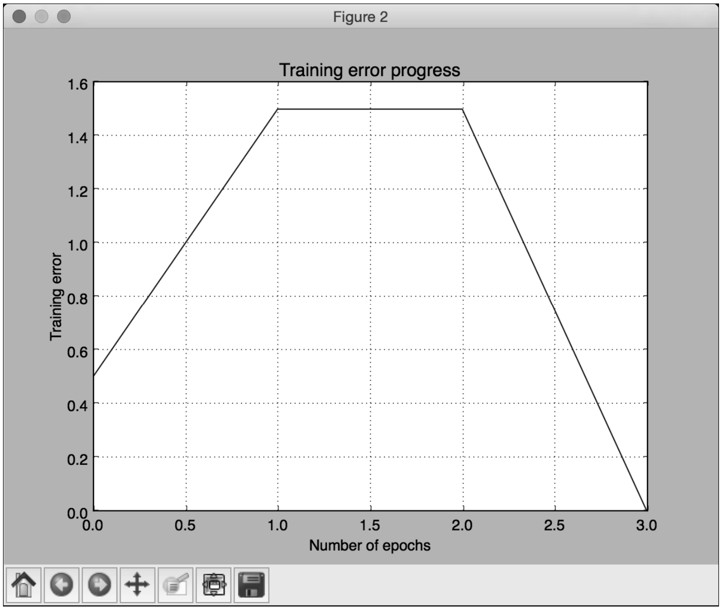


图 11-3

11.3 创建一个单层神经网络

知道如何创建一个感知器后，接下来创建一个单层神经网络。单层神经网络由一个层次中的多个神经元组成。总体上看，单层神经网络将会有一个输入层、一个隐藏层和一个输出层。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

(2) 本例将会用到data_single_layer.txt文件中的数据。先加载这个文件：

```
# 定义输入数据
input_file = 'data_single_layer.txt'
input_text = np.loadtxt(input_file)
data = input_text[:, 0:2]
labels = input_text[:, 2:]
```

(3) 画出输入数据：

```
# 画出输入数据
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Input data')
```

(4) 提取最小值和最大值：

```
# 提取每个维度的最小值和最大值
x_min, x_max = data[:,0].min(), data[:,0].max()
y_min, y_max = data[:,1].min(), data[:,1].max()
```

(5) 定义一个单层神经网络，该神经网络的隐藏层包含两个神经元：

```
# 定义一个单层神经网络，包含两个神经元；在感知器第一个参数的每个元素中指定参数的最大值和最小值
single_layer_net = nl.net.newp([[x_min, x_max], [y_min, y_max]], 2)
```

(6) 通过50次迭代训练该神经网络：

```
# 训练神经网络
error = single_layer_net.train(data, labels, epochs=50, show=20, lr=0.01)
```

(7) 画出结果：

```
# 画出结果
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.title('Training error progress')
plt.grid()

plt.show()
```

(8) 用新的测试数据来测试神经网络：

```
print single_layer_net.sim([[0.3, 4.5]])
print single_layer_net.sim([[4.5, 0.5]])
print single_layer_net.sim([[4.3, 8]])
```

(9) 全部代码已经在single_layer.py文件中给出。运行该代码，可以看到两幅图像。第一幅图像显示输入数据，如图11-4所示。

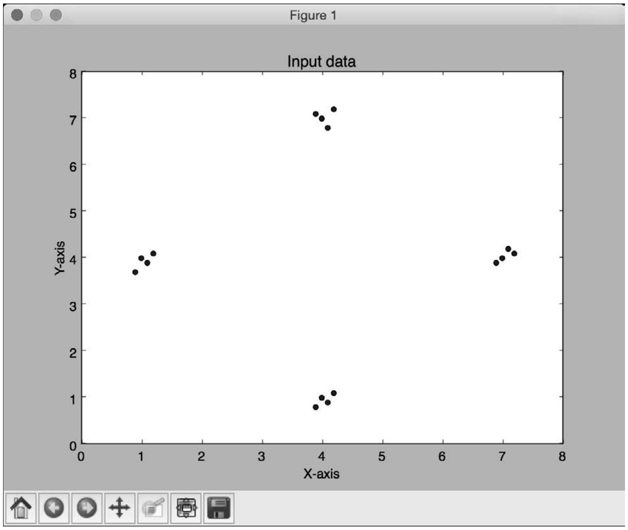


图 11-4

(10) 第二幅图像显示训练误差进程，如图11-5所示。

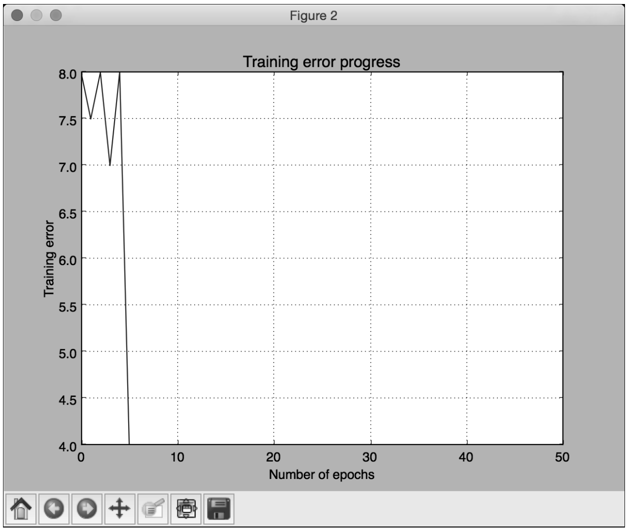


图 11-5

可以在命令行工具中看到如下输出结果，指示输入测试点所属的位置：


```
[[ 0.   0.]]
[[ 1.   0.]]
[[ 1.   1.]]
```

可以基于标签来验证输出结果的准确性。

11.4　创建一个深度神经网络

接下来创建一个深度神经网络。一个深度神经网络由一个输入层、多个隐藏层和一个输出层组成。该模型如图11-6所示。

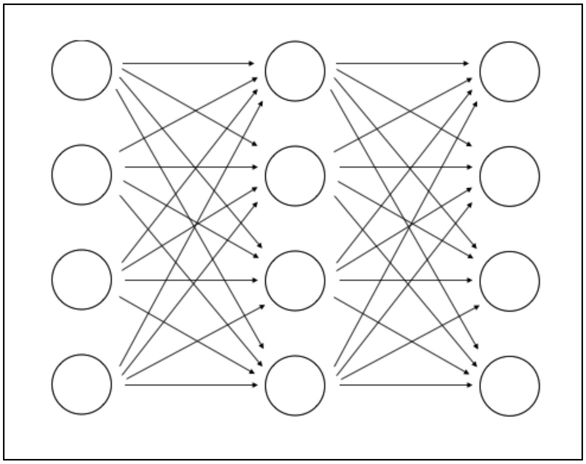


图　11-6

图11-6展示了一个多层神经网络，该神经网络包括一个输入层、多个隐藏层和一个输出层。在一个深度神经网络中，输入层和输出层之间有多个隐藏层。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import neurolab as nl
import numpy as np
import matplotlib.pyplot as plt
```

(2) 定义以下参数，用于生成训练数据：

```
# 生成训练数据
min_value = -12
max_value = 12
num_datapoints = 90
```

(3) 训练数据将由我们定义的一个函数组成，该函数将转换值。我们期望神经网络可以根据提供的输入数据和输出数据来学习这一点：

```
x = np.linspace(min_value, max_value, num_datapoints)
y = 2 * np.square(x) + 7
y /= np.linalg.norm(y)
```

(4) 数组变形：

```
data = x.reshape(num_datapoints, 1)
labels = y.reshape(num_datapoints, 1)
```

(5) 画出输入数据：

```
# 画出输入数据
plt.figure()
plt.scatter(data, labels)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Input data')
```

(6) 定义一个深度神经网络，该神经网络包含两个隐藏层，每个隐藏层包含10个神经元：

```
# 定义一个深度神经网络，带两个隐藏层；每个隐藏层由10个神经元组成，输出层由一个神经元组成
multilayer_net = nl.net.newff([[min_value, max_value]], [10, 10, 1])
```

(7) 设置训练算法为**梯度下降法**（关于梯度下降法的介绍可参考https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression）：

```
# 设置训练算法为梯度下降法
multilayer_net.trainf = nl.train.train_gd
```

(8) 训练网络：

```
# 训练网络
error = multilayer_net.train(data, labels, epochs=800, show=100, goal=0.01)
```

(9) 用训练数据运行该网络，查看其性能表现：

```
# 用训练数据运行该网络，预测结果
predicted_output = multilayer_net.sim(data)
```

(10) 画出训练误差结果：

```
# 画出训练误差结果
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Error')
plt.title('Training error progress')
```

(11) 创建一组新的输入数据，并运行神经网络，查看其性能表现：

```
# 画出预测结果
x2 = np.linspace(min_value, max_value, num_datapoints * 2)
y2 = multilayer_net.sim(x2.reshape(x2.size,1)).reshape(x2.size)
y3 = predicted_output.reshape(num_datapoints)
```


(12) 画出输出结果：

```
plt.figure()
plt.plot(x2, y2, '-', x, y, '.', x, y3, 'p')
```

```
plt.title('Ground truth vs predicted output')

plt.show()
```

(13) 全部代码已经在deep_neural_network.py文件中给出。运行该代码，可以看到3幅图像。第一幅图像显示输入数据，如图11-7所示。

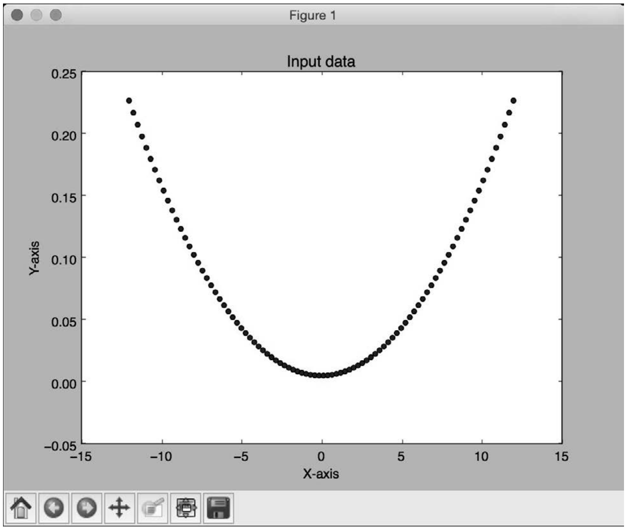


图　11-7

第二幅图像显示训练误差进程，如图11-8所示。

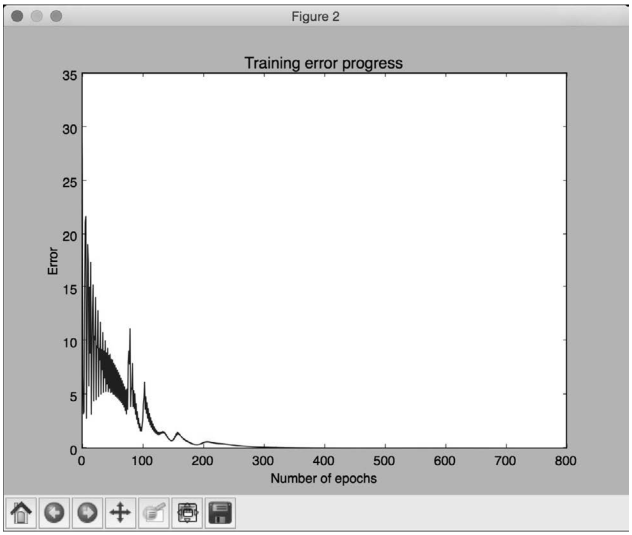


图　11-8

第三幅图像显示神经网络的输出，如图11-9所示。

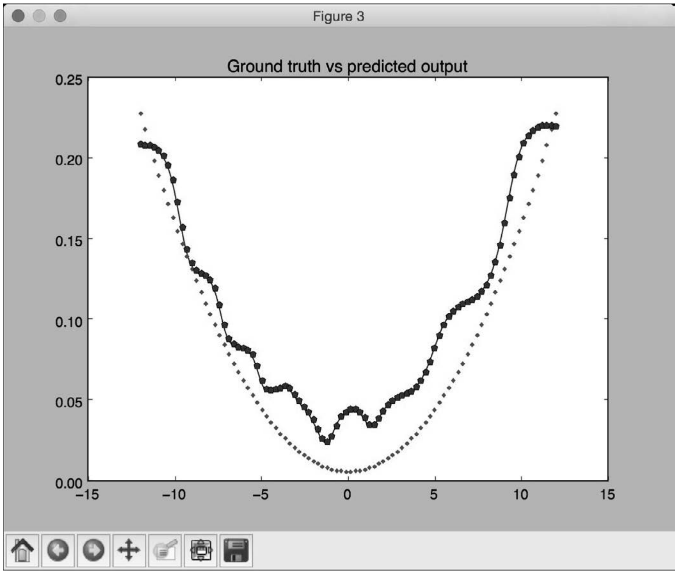


图　11-9

可以在命令行工具中看到如图11-10所示的显示结果。

```
Epoch: 100; Error: 1.64795788647;
Epoch: 200; Error: 0.517736068801;
Epoch: 300; Error: 0.13545620002;
Epoch: 400; Error: 0.0521272422892;
Epoch: 500; Error: 0.0465021594702;
Epoch: 600; Error: 0.0483261849312;
Epoch: 700; Error: 0.0431681554217;
Epoch: 800; Error: 0.0346446191022;
The maximum number of train epochs is reached
```

图　11-10

11.5　创建一个向量量化器

你也可以用神经网络来做向量量化。**向量量化**是*N*维空间的“四舍五入”，广泛用于各个领域，如计算机视觉、自然语言处理和机器学习等。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

(2) 从data_vq.txt文件加载输入数据：

```
# 定义输入数据
input_file = 'data_vq.txt'
input_text = np.loadtxt(input_file)
data = input_text[:, 0:2]
labels = input_text[:, 2:]
```

(3) 定义一个两层的**学习向量量化**（Learning Vector Quantization，LVQ）神经网络。函数中最后一个参数的数组指定了每个输出的加权百分比（各加权百分比之和应为1）：

```
# 定义一个两层神经网络：输入含10个神经元，输出含4个神经元
net = nl.net.newlvq(nl.tool.minmax(data), 10, [0.25, 0.25, 0.25, 0.25])
```

(4) 训练LVQ神经网络：

```
# 训练神经网络
error = net.train(data, labels, epochs=100, goal=-1)
```

(5) 创建一个用于测试及可视化的网格点值：

```
# 创建输入网格
xx, yy = np.meshgrid(np.arange(0, 8, 0.2), np.arange(0, 8, 0.2))
xx.shape = xx.size, 1
yy.shape = yy.size, 1
input_grid = np.concatenate((xx, yy), axis=1)
```

(6) 用这些网格点值评价该网络：

```
# 用这些网格点值评价该网络
output_grid = net.sim(input_grid)
```

(7) 在数据中定义4个类：

```
# 定义4个类
class1 = data[labels[:,0] == 1]
class2 = data[labels[:,1] == 1]
class3 = data[labels[:,2] == 1]
class4 = data[labels[:,3] == 1]
```

(8) 为每个类定义网格：

```
# 为4个类定义网格
grid1 = input_grid[output_grid[:,0] == 1]
grid2 = input_grid[output_grid[:,1] == 1]
grid3 = input_grid[output_grid[:,2] == 1]
grid4 = input_grid[output_grid[:,3] == 1]
```

(9) 画出输出结果：

```
# 画出输出结果
plt.plot(class1[:,0], class1[:,1], 'ko', class2[:,0], class2[:,1], 'ko',
                class3[:,0], class3[:,1], 'ko', class4[:,0], class4[:,1], 'ko')
plt.plot(grid1[:,0], grid1[:,1], 'b.', grid2[:,0], grid2[:,1], 'gx',
                grid3[:,0], grid3[:,1], 'cs', grid4[:,0], grid4[:,1], 'ro')
plt.axis([0, 8, 0, 8])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Vector quantization using neural networks')

plt.show()
```

(10) 全部代码已经在vector_quantization.py文件中给出。运行该代码，可以看到图像空间被分成不同区域，如图11-11所示。每个区域对应空间中向量量化区域列表中的一个。

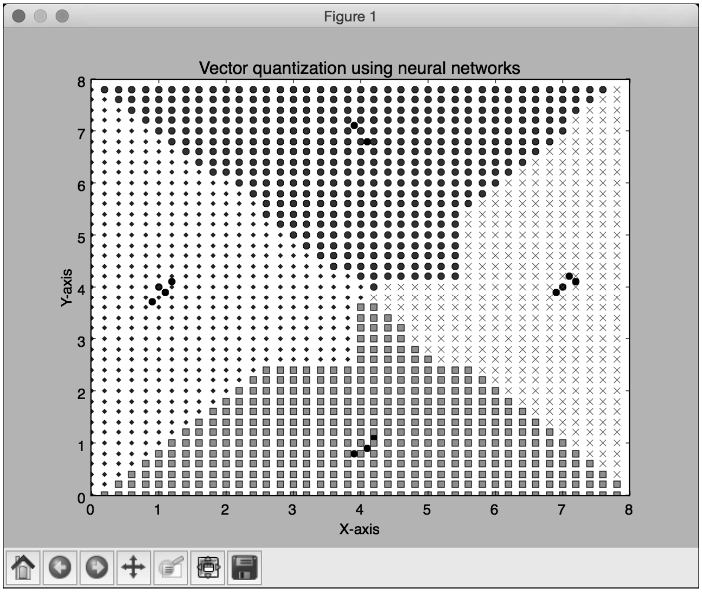


图　11-11

11.6　为序列数据分析创建一个递归神经网络

递归神经网络能较好地分析序列和时间序列数据。你可以在http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns看到更多关于递归神经网络的详细内容。在处理序列和时间序列数据时，不能简单地扩展通用模型。数据的时序相关性非常关键，构建模型时需要考虑到这一点。接下来看看如何创建递归神经网络。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

(2) 定义一个函数，该函数利用输入参数创建一个波形：

```
def create_waveform(num_points):
    # 创建训练样本
    data1 = 1 * np.cos(np.arange(0, num_points))
    data2 = 2 * np.cos(np.arange(0, num_points))
    data3 = 3 * np.cos(np.arange(0, num_points))
    data4 = 4 * np.cos(np.arange(0, num_points))
```

(3) 为每个区间创建不同的振幅，以此来创建一个随机波形：

```
    # 创建不同的振幅
    amp1 = np.ones(num_points)
    amp2 = 4 + np.zeros(num_points)
    amp3 = 2 * np.ones(num_points)
    amp4 = 0.5 + np.zeros(num_points)
```

(4) 将数组合并生成输出数组，其中数据对应输入，而振幅对应相应的标签：

```
    data = np.array([data1, data2, data3, data4]).reshape(num_points * 4, 1)
    amplitude = np.array([[amp1, amp2, amp3, amp4]]).reshape(num_points * 4, 1)

    return data, amplitude
```

(5) 定义一个函数，用于画出将数据传入训练的神经网络后的输出：

```
# 使用网络画出输出结果
def draw_output(net, num_points_test):
    data_test, amplitude_test = create_waveform(num_points_test)
    output_test = net.sim(data_test)
    plt.plot(amplitude_test.reshape(num_points_test * 4))
    plt.plot(output_test.reshape(num_points_test * 4))
```

(6) 定义main函数，并生成示例数据：

```
if __name__=='__main__' :
    # 获取数据
    num_points = 30
    data, amplitude = create_waveform(num_points)
```

(7) 创建一个两层的递归神经网络：

```
    # 创建一个两层的神经网络
    net = nl.net.newelm([[-2, 2]], [10, 1], [nl.trans.TanSig(), nl.trans.PureLin()])
```

(8) 设定每层的初始化函数：

```
# 设定初始化函数并进行初始化
net.layers[0].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
net.layers[1].initf= nl.init.InitRand([-0.1, 0.1], 'wb')
net.init()
```

(9) 训练递归神经网络：

```
# 训练递归神经网络
error = net.train(data, amplitude, epochs=1000, show=100, goal=0.01)
```

(10) 为训练数据计算来自网络的输出：

```
# 计算来自网络的输出
output = net.sim(data)
```

(11) 画出训练误差：

```
# 画出训练结果
plt.subplot(211)
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Error (MSE)')
```

(12) 画出结果：

```
plt.subplot(212)
plt.plot(amplitude.reshape(num_points * 4))
plt.plot(output.reshape(num_points * 4))
plt.legend(['Ground truth', 'Predicted output'])
```

(13) 创建一个随机长度的波形，查看该神经网络能否预测：

```
# 在多个尺度上对未知数据进行测试
plt.figure()

plt.subplot(211)
draw_output(net, 74)
plt.xlim([0, 300])
```

(14) 创建另一个长度更短的波形，查看该神经网络能否预测：

```
plt.subplot(212)
draw_output(net, 54)
plt.xlim([0, 300])

plt.show()
```


(15) 全部代码已经在recurrent_network.py文件中给出。运行该代码，可以看到两幅图像。第一幅图像展示的是训练数据的训练误差及其性能表现，如图11-12所示。

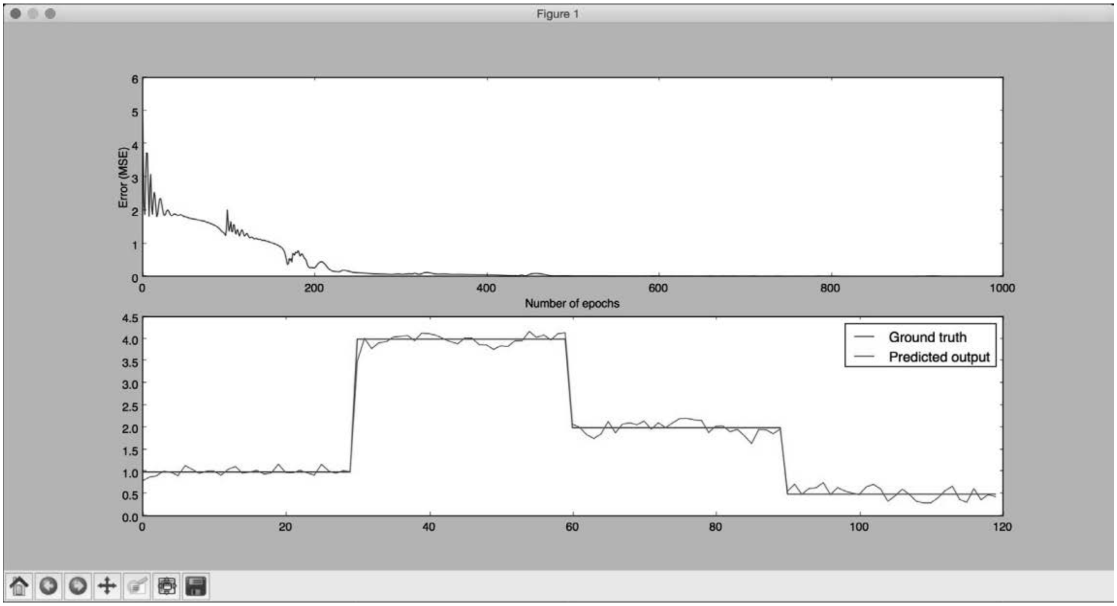


图 11-12

第二幅图像展示的是该训练过的神经网络对于任意长度序列的表现，如图11-13所示。

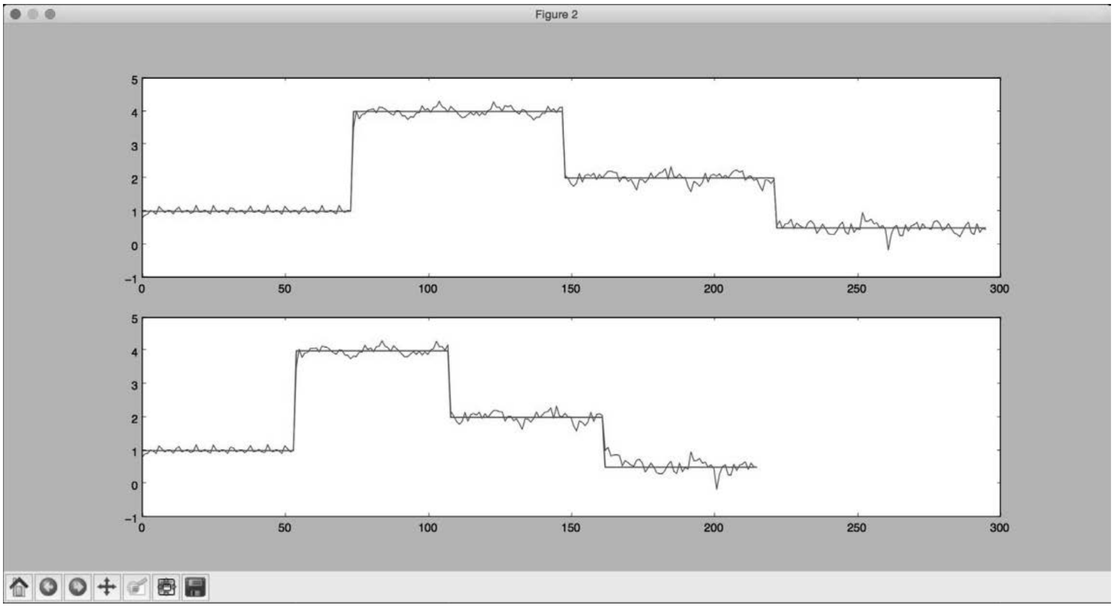


图 11-13

可以在命令行工具中看到如图11-14所示的结果。

```
Epoch: 100; Error: 2.0202165367;
Epoch: 200; Error: 0.276370891;
Epoch: 300; Error: 0.0902055024828;
Epoch: 400; Error: 0.0662254210369;
Epoch: 500; Error: 0.0291456739963;
Epoch: 600; Error: 0.0274479103273;
Epoch: 700; Error: 0.0221256973779;
Epoch: 800; Error: 0.0227723305931;
Epoch: 900; Error: 0.0207200477057;
Epoch: 1000; Error: 0.0159299080472;
The maximum number of train epochs is reached
```

图 11-14

11.7 在光学字符识别数据库中将字符可视化

接下来看看如何利用神经网络做光学字符的识别。光学字符识别是指识别图像中的手写字符的过程。我们会用到http://ai.stanford.edu/~btaskar/ocr中提供的数据集，下载后的默认文件名为letter.data。首先来看如何处理这些数据并将其可视化。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import os
import sys

import cv2
import numpy as np
```

(2) 定义输入文件名：

```
# 加载数据
input_file = 'letter.data'
```

(3) 定义可视化参数：

```
# 定义可视化参数
scaling_factor = 10
start_index = 6
end_index = -1
h, w = 16, 8
```


(4) 循环迭代文件直至用户按下Esc键。用Tab分隔符将行分隔成字符：

```
# 循环直至用户按下Esc键
with open(input_file, 'r') as f:
    for line in f.readlines():
```

```
        data = np.array([255*float(x) for x in line.split('\t')
              [start_index:end_index]])
```

(5) 将数组重新调整为所需的形状，调整大小并将其展示：

```
        img = np.reshape(data, (h,w))
        img_scaled = cv2.resize(img, None, fx=scaling_factor, fy=scaling_factor)
        cv2.imshow('Image', img_scaled)
```

(6) 如果用户按下Esc键，则终止循环：

```
        c = cv2.waitKey()
        if c == 27:
            break
```

(7) 全部代码已经在visualize_characters.py文件中给出。运行该代码，可以看到一个展示字符的窗体。例如，字母“o”的形状如图11-15所示。

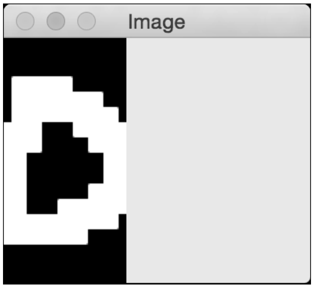


图　11-15

字母“i”的形状如图11-16所示。

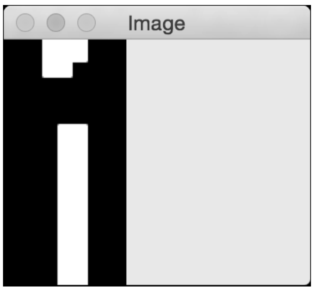


图　11-16

11.8　用神经网络创建一个光学字符识别器

知道如何与数据交互后，接下来创建一个基于神经网络的光学字符识别系统。

详细步骤

(1) 创建一个Python文件，并导入以下程序包：

```
import numpy as np
import neurolab as nl
```

(2) 定义输入文件名称：

```
# 输入文件
input_file = 'letter.data'
```

(3) 在用神经网络处理大量数据时，往往需要花费很多时间来做训练。为了展示如何创建这个系统，这里只使用20个数据点：

```
# 从输入文件加载数据点
num_datapoints = 20
```

(4) 观察数据，可以看到在前20行有7个不同的字符。将其定义如下：

```
# 不同的字符
orig_labels = 'omandig'

# 不同字符的数量
num_output = len(orig_labels)
```

(5) 用数据集的90%做训练，剩下的10%做测试。定义训练和测试参数如下：

```
# 定义训练和测试参数
num_train = int(0.9 * num_datapoints)
num_test = num_datapoints - num_train
```

(6) 数据文件中每行的起始索引值和终止索引值设置如下：

```
# 定义数据集提取参数
start_index = 6
end_index = -1
```

(7) 生成数据集：

```
# 生成数据集
data = []
labels = []
with open(input_file, 'r') as f:
    for line in f.readlines():
        # 按Tab键分割
        list_vals = line.split('\t')
```


(8) 增加一个错误检查步骤，以查看这些字符是否在标签列表中：

```
        # 如果字符不在标签列表中，跳过
        if list_vals[1] not in orig_labels:
            continue
```

(9) 提取标签，并将其添加到主列表的后面：

```
# 提取标签，并将其添加到主列表的后面
label = np.zeros((num_output, 1))
label[orig_labels.index(list_vals[1])] = 1
labels.append(label)
```

(10) 提取字符，并将其添加到主列表的后面：

```
# 提取字符，并将其添加到主列表的后面
cur_char = np.array([float(x) for x in list_vals[start_index:end_index]])
data.append(cur_char)
```

(11) 当有足够多数据时跳出循环：

```
# 当有足够多数据时跳出循环
if len(data) >= num_datapoints:
    break
```

(12) 将以上数据转换成NumPy数组：

```
# 将数据转换成NumPy数组
data = np.asfarray(data)
labels = np.array(labels).reshape(num_datapoints, num_output)
```

(13) 提取数据的维度信息：

```
# 提取数据的维度信息
num_dims = len(data[0])
```

(14) 用10 000次迭代来训练神经网络：

```
# 创建并训练神经网络
net = nl.net.newff([[0, 1] for _ in range(len(data[0]))], [128, 16, num_output])
net.trainf = nl.train.train_gd
error = net.train(data[:num_train,:], labels[:num_train,:], epochs=10000,
show=100, goal=0.01)
```

(15) 为测试输入数据预测输出结构：

```
# 为测试输入数据预测输出结构
predicted_output = net.sim(data[num_train:, :])
print "\nTesting on unknown data:"
for i in range(num_test):
    print "\nOriginal:", orig_labels[np.argmax(labels[i])]
    print "Predicted:", orig_labels[np.argmax(predicted_output[i])]
```

(16) 全部代码已经在ocr.py文件中给出。运行该代码，可以在命令行工具中看到如图11-17所示的结果。

```
Epoch: 8900; Error: 0.167967242056;
Epoch: 9000; Error: 0.141695146517;
Epoch: 9100; Error: 0.123315386106;
Epoch: 9200; Error: 0.123516370118;
Epoch: 9300; Error: 0.153659730139;
Epoch: 9400; Error: 0.106912207871;
Epoch: 9500; Error: 0.0833274962321;
Epoch: 9600; Error: 0.204787347758;
Epoch: 9700; Error: 0.208864612943;
Epoch: 9800; Error: 0.177338615833;
Epoch: 9900; Error: 0.152109654098;
Epoch: 10000; Error: 0.130557716464;
The maximum number of train epochs is reached
```

图 11-17

神经网络的输出结果如图11-18所示。

```
Testing on unknown data:

Original: o
Predicted: o

Original: m
Predicted: m
```

图 11-18

第 12 章

可视化数据

12

在这一章，我们将介绍以下主题：

- 画3D散点图
- 画气泡图
- 画动态气泡图
- 画饼图
- 画日期格式的时间序列数据
- 画直方图
- 可视化热力图
- 动态信号的可视化模拟

12.1　简介

数据可视化是机器学习的核心，利用它有助于制定正确的策略来理解数据。数据的视觉表示帮助我们选择正确的算法。数据可视化的主要目标之一就是用图和表清晰地表达出数据，以便我们更准确、更有效地交流信息。

在现实世界中总会存在各种数值数据，我们想将这些数值数据编码成图、线、点、条等，以便直观地显示这些数值中包含的信息，同时可以使复杂分布的数据更容易被理解和应用。这一过程被广泛应用于各种场合之中，包括对比分析、增长率跟踪、市场分布、民意调查等。

我们用不同的图来展示各个变量之间的模式或关系，比如用直方图展示数据的分布。如果想查找一个特定的测量，可以用表格表示。这一章将讨论各种场景下最合适的可视化方式。

12.2　画 3D 散点图

这一节将学习如何画3D散点图，并学习如何在三维空间中可视化这些点。

详细步骤

(1) 生成一个新的Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

(2) 生成一个空白图像：

```
# 生成一个空白图像
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

(3) 定义应该生成的值的个数：

```
# 定义生成的值的个数
n = 250
```

(4) 生成一个`lambda`函数来生成给定范围的值：

```
# 生成lambda函数来生成给定范围的值
f = lambda minval, maxval, n: minval + (maxval - minval) * np.random.rand(n)
```

(5) 用这个函数生成X、Y和Z值：

```
# 生成值
x_vals = f(15, 41, n)
y_vals = f(-10, 70, n)
z_vals = f(-52, -37, n)
```

(6) 画出这些值：

```
# 画出这些值
ax.scatter(x_vals, y_vals, z_vals, c='k', marker='o')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

plt.show()
```

(7) 全部代码已经包含在scatter_3d.py文件中。运行该代码，可以看到如图12-1所示的图像。

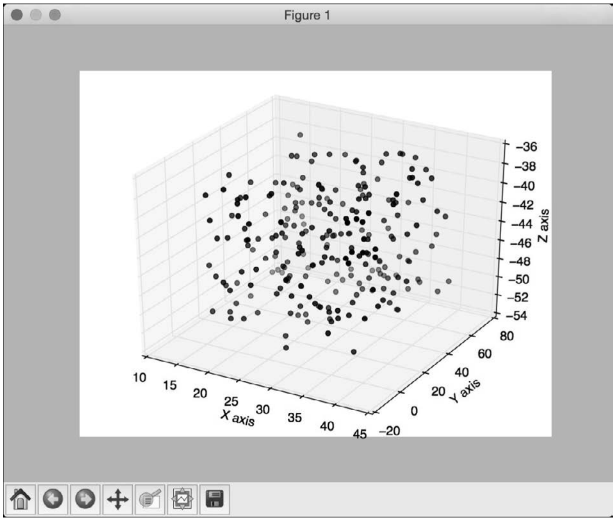


图　12-1

12.3　画气泡图

下面看看如何画气泡图。在二维的气泡图中，每一个圆圈的大小表示这个点的幅值。

详细步骤

(1) 生成一个Python文件，并导入如下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
```

(2) 定义要生成的值的个数：

```
# 定义值的个数
num_vals = 40
```

(3) 生成随机的x值和y值：

```
# 生成随机数
x = np.random.rand(num_vals)
y = np.random.rand(num_vals)
```

(4) 在气泡图中定义每个点的面积值：

```
# 定义每个点的面积
# 指定最大半径
max_radius = 25
area = np.pi * (max_radius * np.random.rand(num_vals)) ** 2
```

(5) 定义颜色：

```
# 生成颜色
colors = np.random.rand(num_vals)
```

(6) 画出这些值：

```
# 画出数据点
plt.scatter(x, y, s=area, c=colors, alpha=1.0)

plt.show()
```

(7) 全部代码已经包含在bubble_plot.py文件中。运行该代码，可以看到如图12-2所示的图像。

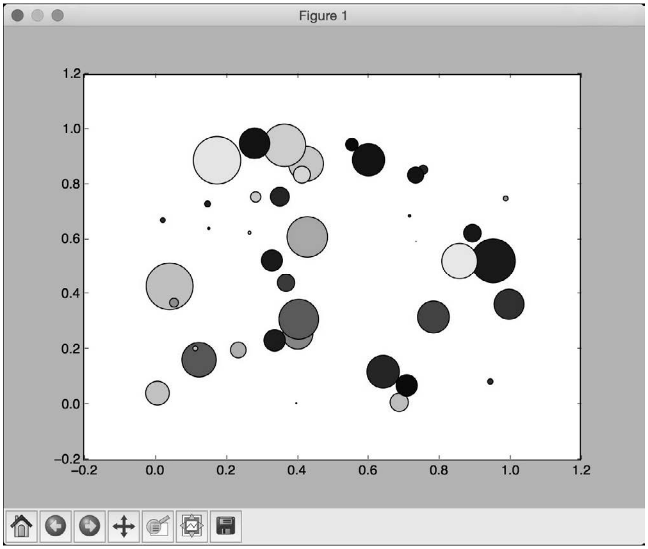


图 12-2

12.4 画动态气泡图

下面来看看如何画动态气泡图。如果需要将动态的数据可视化，那就需要用到该可视化方式。

详细步骤

(1) 生成一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
```

(2) 定义一个tracker函数，该函数将动态更新气泡图：

```
def tracker(cur_num):
    # 获取当前索引
    cur_index = cur_num % num_points
```

(3) 定义颜色：

```
    # 定义数据点颜色
    datapoints['color'][:, 3] = 1.0
```

(4) 更新圆圈的大小：

```
    # 更新圆圈的大小
    datapoints['size'] += datapoints['growth']
```

(5) 更新集合中最老的数据点的位置：

```
    # 更新集合中最老的数据点的位置
    datapoints['position'][cur_index] = np.random.uniform(0, 1, 2)
    datapoints['size'][cur_index] = 7
    datapoints['color'][cur_index] = (0, 0, 0, 1)
    datapoints['growth'][cur_index] = np.random.uniform(40, 150)
```

(6) 更新散点图的参数：

```
    # 更新散点图的参数
    scatter_plot.set_edgecolors(datapoints['color'])
    scatter_plot.set_sizes(datapoints['size'])
    scatter_plot.set_offsets(datapoints['position'])
```

(7) 定义main函数并生成一个空白的图像：

```
if __name__=='__main__':
    # 生成一个图像
    fig = plt.figure(figsize=(9, 7), facecolor=(0,0.9,0.9))
    ax = fig.add_axes([0, 0, 1, 1], frameon=False)
    ax.set_xlim(0, 1), ax.set_xticks([])
    ax.set_ylim(0, 1), ax.set_yticks([])
```

(8) 定义在任意时间点上的点的个数：

```
    # 在随机位置创建和初始化数据点，并以随机的增长率进行初始化
    num_points = 20
```

(9) 用随机值定义这些数据点：

```
    datapoints = np.zeros(num_points, dtype=[('position', float, 2),
            ('size', float, 1), ('growth', float, 1), ('color', float, 4)])
    datapoints['position'] = np.random.uniform(0, 1, (num_points, 2))
    datapoints['growth'] = np.random.uniform(40, 150, num_points)
```

(10) 创建一个散点图，该散点图的每一帧都会更新：

```
# 创建一个每一帧都会更新的散点图
scatter_plot = ax.scatter(datapoints['position'][:, 0], datapoints['position'][:, 1],
                  s=datapoints['size'], lw=0.7, edgecolors=datapoints['color'],
                  facecolors='none')
```

(11) 用tracker函数启动动态模拟：

```
# 用tracker函数启动动态模拟
animation = FuncAnimation(fig, tracker, interval=10)

plt.show()
```

(12) 全部代码已经包含在dynamic_bubble_plot.py文件中。运行该程序，可以看到如图12-3所示的图像。

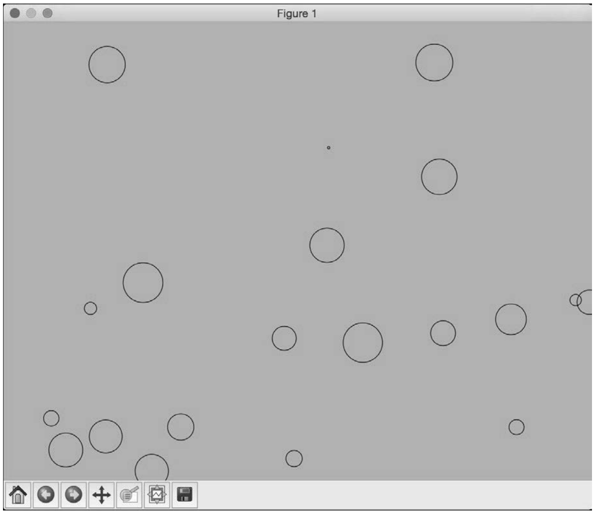


图 12-3

12.5 画饼图

下面来看看如何画一个饼图。如果想将一组数据中各标签的比例值可视化展现时，可以用到该可视化方式。

详细步骤

(1) 生成一个Python文件，并导入以下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
```

(2) 定义各标签和相应的值：

```
# 按照顺时针方向定义各标签和相应的值
data = {'Apple': 26,
        'Mango': 17,
        'Pineapple': 21,
        'Banana': 29,
        'Strawberry': 11}
```

(3) 定义可视化的颜色：

```
# 定义可视化的颜色
colors = ['orange', 'lightgreen', 'lightblue', 'gold', 'cyan']
```

(4) 定义一个变量，以突出饼图的一部分，将其与其他部分分离开。如果不想突出任何部分，将所有值设置为`0`：

```
# 定义是否需要突出一部分
explode = (0, 0, 0, 0, 0)
```

(5) 画饼图。注意，如果使用Python 3版本，应该在下面的函数中调用`list(data.values())`：

```
# 画饼图
plt.pie(data.values(), explode=explode, labels=data.keys(),
        colors=colors, autopct='%1.1f%%', shadow=False, startangle=90)

# 设置饼图的宽高比，“equal”表示我们希望它是圆形的
plt.axis('equal')

plt.show()
```

(6) 全部的代码已经包含在pie_chart.py文件中。运行该代码，可以看到如图12-4所示的图像。

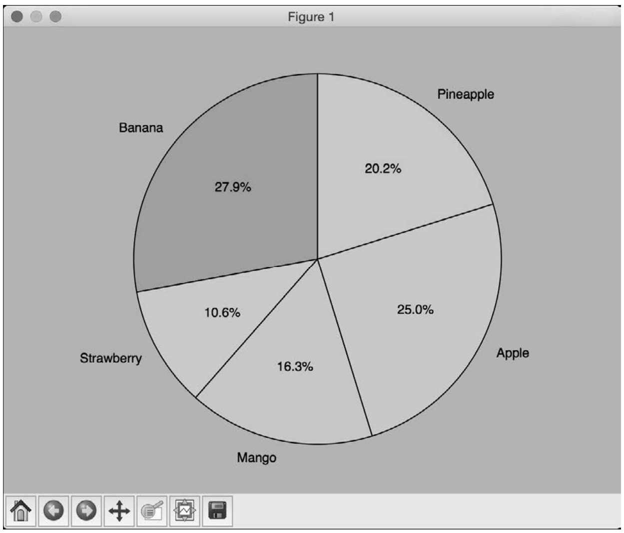


图　12-4

如果将explode数组设置为(0, 0.2, 0, 0, 0)，那么Strawberry部分将分离突出显示，如图12-5所示。

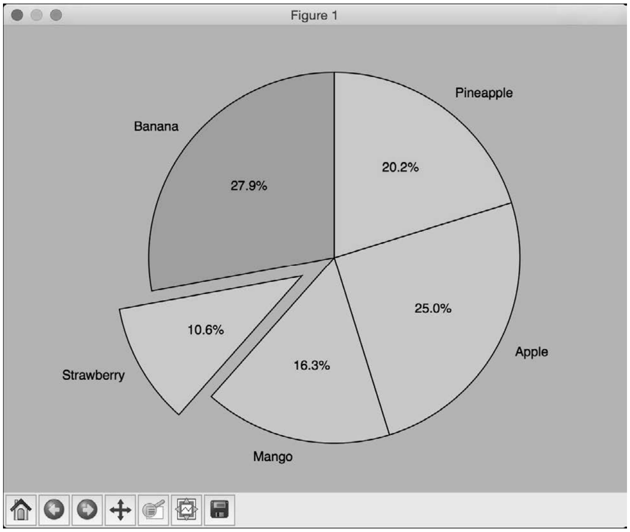


图　12-5

12.6　画日期格式的时间序列数据

下面来看看如何用日期格式画时间序列数据。如果要将各时期的股票数据进行可视化展现，可以使用该可视化形式。

详细步骤

(1) 生成一个Python文件，并导入如下程序包：

```
import numpy
import matplotlib.pyplot as plt
from matplotlib.mlab import csv2rec
import matplotlib.cbook as cbook
from matplotlib.ticker import Formatter
```

(2) 定义一个用于将日期格式化的类。`init`函数设置类变量：

```
# 定义一个类将日期格式化
class DataFormatter(Formatter):
    def __init__(self, dates, date_format='%Y-%m-%d'):
        self.dates = dates
        self.date_format = date_format
```

(3) 提取给定时间的值，并用如下格式将其返回：

```
# 提取“position”位置的时间t的值
def __call__(self, t, position=0):
    index = int(round(t))
    if index >= len(self.dates) or index < 0:
        return ''

    return self.dates[index].strftime(self.date_format)
```

(4) 定义main函数。我们将用到matplotlib中苹果公司的股票报价CSV文件：

```
if __name__=='__main__':
    # 输入包含股价的CSV文件
    input_file = cbook.get_sample_data('aapl.csv', asfileobj=False)
```

(5) 加载CSV文件：

```
    # 将CSV文件加载到numpy记录数组中
    data = csv2rec(input_file)
```

(6) 提取这些值的子集，并将其画出：

```
    # 提取子集并画出
    data = data[-70:]
```

(7) 创建一个格式化对象，并将其用日期数据初始化：

```
    # 创建一个日期格式化对象
    formatter = DataFormatter(data.date)
```

(8) 定义X轴和Y轴：

```
    # X轴
    x_vals = numpy.arange(len(data))

    # Y轴表示收盘价
    y_vals = data.close
```

(9) 画出数据：

```
    # 画出数据
    fig, ax = plt.subplots()
    ax.xaxis.set_major_formatter(formatter)
    ax.plot(x_vals, y_vals, 'o-')
    fig.autofmt_xdate()
    plt.show()
```

(10) 全部的代码已经包含在time_series.py文件中。运行该代码，可以看到如图12-6所示的图像。

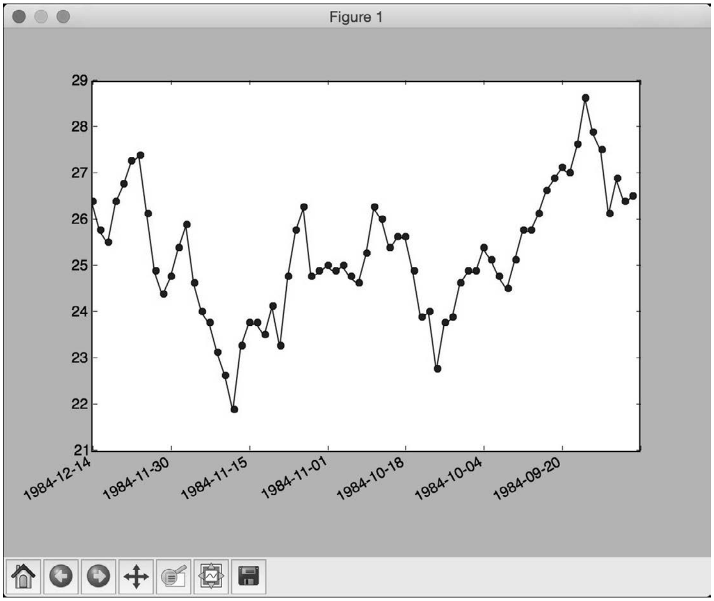


图　12-6

12.7　画直方图

下面来看看如何画直方图。如果需要对比两组数据，可以用直方图做对比。

详细步骤

(1) 生成一个Python文件，并导入如下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
```

(2) 本例对比苹果和橘子的产量，定义如下数据：

```
# 输入数据
apples = [30, 25, 22, 36, 21, 29]
oranges = [24, 33, 19, 27, 35, 20]

# 设置组数
num_groups = len(apples)
```

(3) 创建一个图像并定义其参数：

```
# 创建图像
fig, ax = plt.subplots()
```

```
# 定义X轴
indices = np.arange(num_groups)

# 直方图的宽度和透明度
bar_width = 0.4
opacity = 0.6
```

(4) 画直方图：

```
# 画直方图
hist_apples = plt.bar(indices, apples, bar_width,
        alpha=opacity, color='g', label='Apples')

hist_oranges = plt.bar(indices + bar_width, oranges, bar_width,
        alpha=opacity, color='b', label='Oranges')
```

(5) 设置直方图的参数：

```
plt.xlabel('Month')
plt.ylabel('Production quantity')
plt.title('Comparing apples and oranges')
plt.xticks(indices + bar_width, ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'))
plt.ylim([0, 45])
plt.legend()
plt.tight_layout()

plt.show()
```

(6) 全部的代码已经包含在histogram.py文件中。运行该代码，可以看到如图12-7所示的图像。

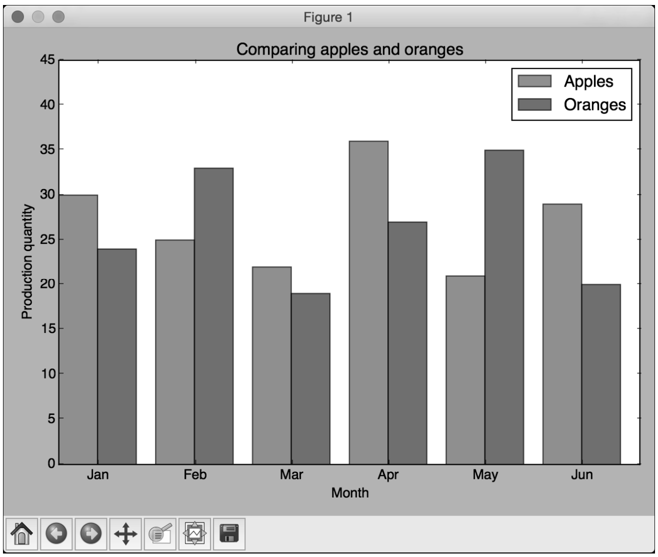


图　12-7

12.8　可视化热力图

接下来看看如何画热力图。当两组数据的各点具有一定的相关性时，可以用这种图形表现方式。矩阵包含的单个值都被表示为图中的颜色值。

详细步骤

(1) 生成一个Python文件，并导入如下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
```

(2) 定义两组数据：

```
# 定义两组数据
group1 = ['France', 'Italy', 'Spain', 'Portugal', 'Germany']
group2 = ['Japan', 'China', 'Brazil', 'Russia', 'Australia']
```

(3) 生成一个随机二维矩阵：

```
# 生成一些随机数
data = np.random.rand(5, 5)
```

(4) 创建一个图像：

```
# 创建一个图像
fig, ax = plt.subplots()
```

(5) 创建一个热力图：

```
# 创建一个热力图
heatmap = ax.pcolor(data, cmap=plt.cm.gray)
```

(6) 画出这些值：

```
# 将坐标轴放在图块的中间
ax.set_xticks(np.arange(data.shape[0]) + 0.5, minor=False)
ax.set_yticks(np.arange(data.shape[1]) + 0.5, minor=False)

# 让热力图显示成一张表
ax.invert_yaxis()
ax.xaxis.tick_top()

# 增加坐标轴标签
ax.set_xticklabels(group2, minor=False)
ax.set_yticklabels(group1, minor=False)

plt.show()
```

(7) 全部的代码已经包含在heatmap.py文件中。运行该代码，可以看到如图12-8所示的图像。

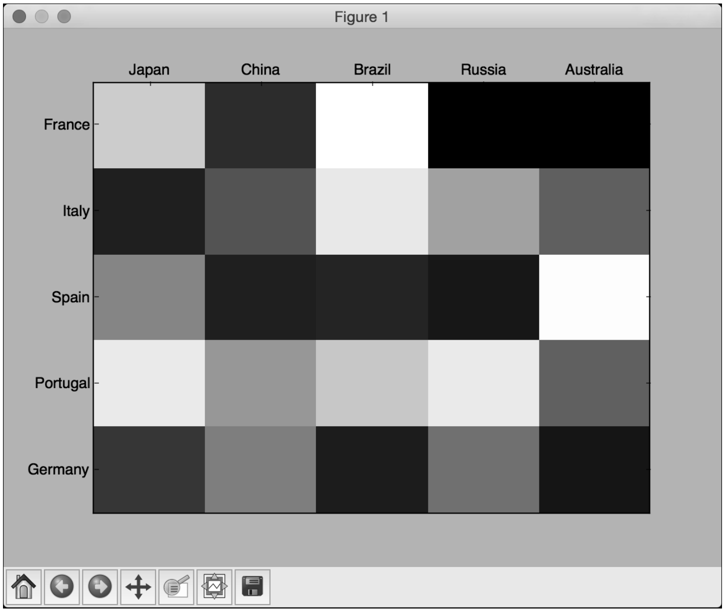


图　12-8

12.9　动态信号的可视化模拟

如果需要将实时信号可视化，最好的方式是看到波形是如何产生的。这一节将讲解如何对实时动态变化的信号进行模拟，并将其可视化。

详细步骤

(1) 生成一个Python文件，并导入如下程序包：

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

(2) 创建一个函数，用于生成阻尼正弦信号：

```
# 生成信号数据
def generate_data(length=2500, t=0, step_size=0.05):
    for count in range(length):
        t += step_size
        signal = np.sin(2*np.pi*t)
        damper = np.exp(-t/8.0)
        yield t, signal * damper
```

(3) 定义一个`initializer`函数，用于将图像中的参数初始化：

```
# 定义初始化函数
def initializer():
    peak_val = 1.0
    buffer_val = 0.1
```

(4) 设置这些参数：

```
    ax.set_ylim(-peak_val * (1 + buffer_val), peak_val * (1 + buffer_val))
    ax.set_xlim(0, 10)
    del x_vals[:]
    del y_vals[:]
    line.set_data(x_vals, y_vals)
    return line
```

(5) 定义一个函数来画出这些值：

```
def draw(data):
    # 升级数据
    t, signal = data
    x_vals.append(t)
    y_vals.append(signal)
    x_min, x_max = ax.get_xlim()
```

(6) 如果这些值超出当前X轴最大值的范围，那么更新X轴最大值并扩展图像：

```
    if t >= x_max:
        ax.set_xlim(x_min, 2 * x_max)
        ax.figure.canvas.draw()

    line.set_data(x_vals, y_vals)

    return line
```

(7) 定义`main`函数：

```
if __name__=='__main__':
    # 创建图形
    fig, ax = plt.subplots()
    ax.grid()
```

(8) 提取线：

```
    # 提取线
    line, = ax.plot([], [], lw=1.5)
```

(9) 创建变量，并用空列表对其初始化：

```
   # 创建变量
    x_vals, y_vals = [], []
```

(10) 用动画器对象定义并启动动画：

```
# 定义动画器对象
animator = animation.FuncAnimation(fig, draw, generate_data,
        blit=False, interval=10, repeat=False, init_func=initializer)

plt.show()
```

(11) 全部的代码已经包含在moving_wave_variable.py文件中。运行该代码，可以看到如图12-9所示的图像。

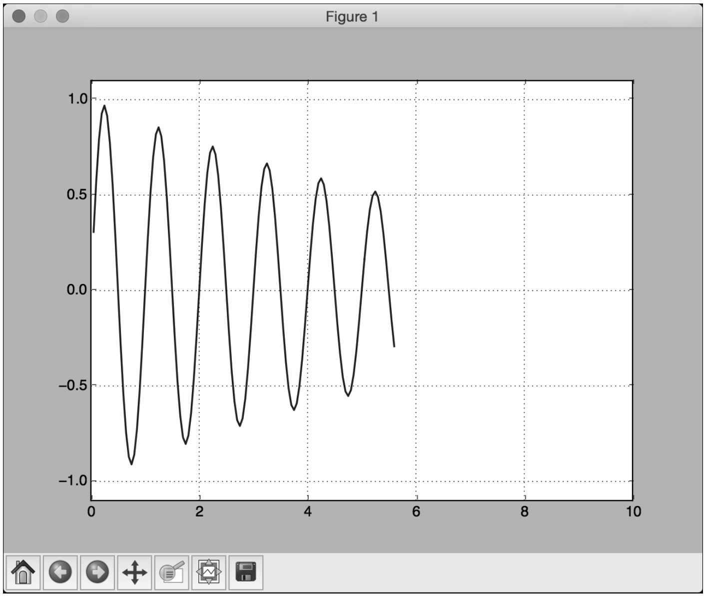


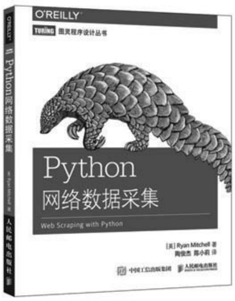
图　12-9

延 展 阅 读



- PSF研究员、知名PyCon演讲者心血之作，Python核心开发人员担纲技术审校
- 全面深入，对Python语言关键特性剖析到位
- 大量详尽代码示例，并附有主题相关高质量参考文献和视频链接
- 兼顾Python 3和Python 2

书号：978-7-115-45415-7
定价：139.00 元



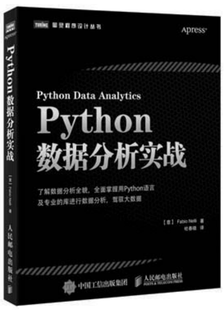
采用简洁强大的Python语言，全面介绍网络数据采集技术，教你从不同形式的网络资源中自由地获取数据。你将学会如何使用Python脚本和网络API一次性采集并处理成千上万个网页上的数据。

书号：978-7-115-41629-2
定价：59.00 元



采用基于项目的方法，介绍用Python完成数据获取、数据清洗、数据探索、数据呈现、数据规模化和自动化的过程。

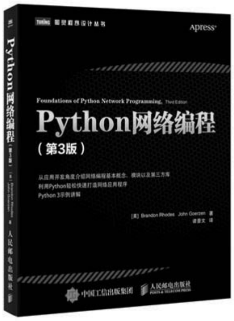
书号：978-7-115-45919-0
定价：99.00 元



了解数据分析全貌，全面掌握用Python语言及专业的库进行数据分析，驾驭大数据。

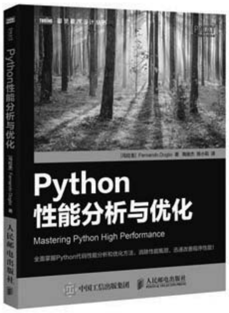
书号：978-7-115-43220-9
定价：59.00 元

延 展 阅 读



- 从应用开发角度介绍网络编程基本概念、模块以及第三方库
- 利用Python轻松快速打造网络应用程序
- Python 3示例讲解

书号：978-7-115-43350-3
定价：79.00 元



全面掌握Python代码性能分析和优化方法，消除性能瓶颈，迅速改善程序性能！

书号：978-7-115-42422-8
定价：45.00 元



- 你一定能看懂的算法基础书
- 代码示例基于Python
- 400多个示意图，生动介绍算法执行过程
- 展示不同算法在性能方面的优缺点
- 教会你用常见算法解决每天面临的实际编程问题

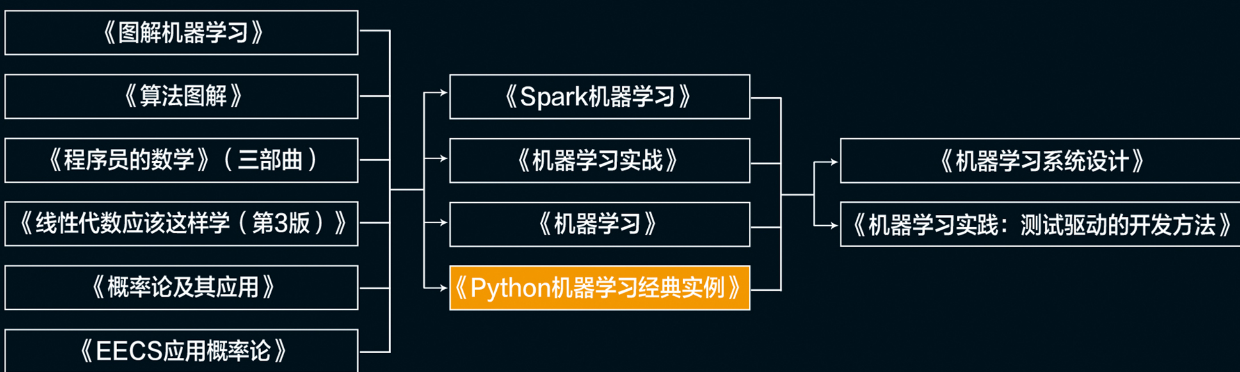
书号：978-7-115-44763-0
定价：49.00 元



- 技术人员版《人性的弱点》
- 提升职业生涯软技能
- 探讨领导力、合作、沟通、高效等团队成功关键因素

书号：978-7-115-43418-0
定价：45.00 元

- 监督学习技术、预测建模、无监督学习算法等前沿话题的实例代码展示
- 来自Kaggle的经典数据集和机器学习案例
- 用流行的Python库scikit-learn解决机器学习问题



[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/程序设计/Python
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46527-6
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks